TOSHIBA

TX System RISC
TX79 Core Architecture

(Symmetric 2-way superscalar 64-bit CPU) Rev. 2.0

TOSHIBA CORPORATION

The information contained herein is subject to change without notice.

The information contained herein is presented only as a guide for the applications of our products. No responsibility is assumed by TOSHIBA for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of TOSHIBA or others.

TOSHIBA is continually working to improve the quality and reliability of its products. Nevertheless, semiconductor devices in general can malfunction or fail due to their inherent electrical sensitivity and vulnerability to physical stress.
It is the responsibility of the buyer, when utilizing TOSHIBA products, to comply with the standards of safety in making a safe design for the entire system, and to avoid situations in which a malfunction or failure of such TOSHIBA products could cause loss of human life, bodily injury or damage to property.
In developing your designs, please ensure that TOSHIBA products are used within specified operating ranges as set forth in the most recent TOSHIBA products specifications.
Also, please keep in mind the precautions and conditions set forth in the “Handling Guide for Semiconductor Devices,” or “TOSHIBA Semiconductor Reliability Handbook” etc..

The Toshiba products listed in this document are intended for usage in general electronics applications (computer, personal equipment, office equipment, measuring equipment, industrial robotics, domestic appliances, etc.).
These Toshiba products are neither intended nor warranted for usage in equipment that requires extraordinarily high quality and/or reliability or a malfunction or failure of which may cause loss of human life or bodily injury (“Unintended Usage”). Unintended Usage include atomic energy control instruments, airplane or spaceship instruments, transportation instruments, traffic signal instruments, combustion control instruments, medical instruments, all types of safety devices, etc.. Unintended Usage of Toshiba products listed in this document shall be made at the customer’s own risk.

The products described in this document may include products subject to the foreign exchange and foreign trade laws.

© 2001 TOSHIBA CORPORATION
All Rights Reserved

Preface

Thank you for choosing Toshiba semiconductor products. This is the year 2000 edition of the user’s manual for the architecture of the TX79 RISC microprocessor core, a member of the TX System RISC Family of Toshiba microprocessors.

This user’s manual is designed to be easily understood by engineers who are designing a Toshiba microprocessor into their products for the first time. No special knowledge of this architecture is assumed – the contents includes basic information about the architecture of the TX79 microprocessor core as well as more advanced, in-depth description.

Toshiba are continually updating technical publications. Any comments and suggestions regarding any Toshiba document are most welcome and will be taken into account when subsequent editions are prepared. To receive updates to the information in this manual, or for additional information about this architecture, please contact your nearest Toshiba office or authorized Toshiba dealer.

April 2001

CONTENTS

FIGURES

TABLES

Handling Precautions

1. Using Toshiba Semiconductors Safely

TOSHIBA is continually working to improve the quality and the reliability of its products.

Nevertheless, semiconductor devices in general can malfunction or fail due to their inherent electrical sensitivity and vulnerability to physical stress. It is the responsibility of the buyer, when utilizing TOSHIBA products, to observe standards of safety, and to avoid situations in which a malfunction or failure of a TOSHIBA product could cause loss of human life, bodily injury or damage to property.

In developing your designs, please ensure that TOSHIBA products are used within specified operating ranges as set forth in the most recent products specifications. Also, please keep in mind the precautions and conditions set forth in the TOSHIBA Semiconductor Reliability Handbook.

2. Safety Precautions

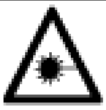
This section lists important precautions which users of semiconductor devices (and anyone else) should observe in order to avoid injury and damage to property, and to ensure safe and correct use of devices.

Please be sure that you understand the meanings of the labels and the graphic symbol described below before you move on to the detailed descriptions of the precautions.

[Explanation of labels]

⚠ DANGER	Indicates an imminently hazardous situation which will result in death or serious injury if you do not follow instructions.
⚠ WARNING	Indicates a potentially hazardous situation which could result in death or serious injury if you do not follow instructions.
⚠ CAUTION	Indicates a potentially hazardous situation which if not avoided, may result in minor injury or moderate injury.

[Explanation of graphic symbol]

Graphic symbol	Meaning
	Indicates that caution is required (laser beam is dangerous to eyes).

2.1 General Precautions regarding Semiconductor Devices

⚠CAUTION

Do not use devices under conditions exceeding their absolute maximum ratings (e.g. current, voltage, power dissipation or temperature).
This may cause the device to break down, degrade its performance, or cause it to catch fire or explode resulting in injury.

Do not insert devices in the wrong orientation.
Make sure that the positive and negative terminals of power supplies are connected correctly. Otherwise the rated maximum current or power dissipation may be exceeded and the device may break down or undergo performance degradation, causing it to catch fire or explode and resulting in injury.

When power to a device is on, do not touch the device's heat sink.
Heat sinks become hot, so you may burn your hand.

Do not touch the tips of device leads.
Because some types of device have leads with pointed tips, you may prick your finger.

When conducting any kind of evaluation, inspection or testing, be sure to connect the testing equipment's electrodes or probes to the pins of the device under test before powering it on.
Otherwise, you may receive an electric shock causing injury.

Before grounding an item of measuring equipment or a soldering iron, check that there is no electrical leakage from it.
Electrical leakage may cause the device which you are testing or soldering to break down, or could give you an electric shock.

Always wear protective glasses when cutting the leads of a device with clippers or a similar tool.
If you do not, small bits of metal flying off the cut ends may damage your eyes.

2.2 Precautions Specific to Each Product Group

2.2.1 Optical semiconductor devices

⚠ DANGER
When a visible semiconductor laser is operating, do not look directly into the laser beam or look through the optical system. This is highly likely to impair vision, and in the worst case may cause blindness. If it is necessary to examine the laser apparatus, for example to inspect its optical characteristics, always wear the appropriate type of laser protective glasses as stipulated by IEC standard IEC825-1.

⚠ WARNING
Ensure that the current flowing in an LED device does not exceed the device's maximum rated current. This is particularly important for resin-packaged LED devices, as excessive current may cause the package resin to blow up, scattering resin fragments and causing injury. When testing the dielectric strength of a photocoupler, use testing equipment which can shut off the supply voltage to the photocoupler. If you detect a leakage current of more than 100 μA, use the testing equipment to shut off the photocoupler's supply voltage; otherwise a large short-circuit current will flow continuously, and the device may break down or burst into flames, resulting in fire or injury. When incorporating a visible semiconductor laser into a design, use the device's internal photodetector or a separate photodetector to stabilize the laser's radiant power so as to ensure that laser beams exceeding the laser's rated radiant power cannot be emitted. If this stabilizing mechanism does not work and the rated radiant power is exceeded, the device may break down or the excessively powerful laser beams may cause injury.

2.2.2 Power devices

⚠ DANGER
Never touch a power device while it is powered on. Also, after turning off a power device, do not touch it until it has thoroughly discharged all remaining electrical charge. Touching a power device while it is powered on or still charged could cause a severe electric shock, resulting in death or serious injury. When conducting any kind of evaluation, inspection or testing, be sure to connect the testing equipment's electrodes or probes to the device under test before powering it on. When you have finished, discharge any electrical charge remaining in the device. Connecting the electrodes or probes of testing equipment to a device while it is powered on may result in electric shock, causing injury.

⚠WARNING

Do not use devices under conditions which exceed their absolute maximum ratings (current, voltage, power dissipation, temperature etc.).
This may cause the device to break down, causing a large short-circuit current to flow, which may in turn cause it to catch fire or explode, resulting in fire or injury.

Use a unit which can detect short-circuit currents and which will shut off the power supply if a short-circuit occurs.
If the power supply is not shut off, a large short-circuit current will flow continuously, which may in turn cause the device to catch fire or explode, resulting in fire or injury.

When designing a case for enclosing your system, consider how best to protect the user from shrapnel in the event of the device catching fire or exploding.
Flying shrapnel can cause injury.

When conducting any kind of evaluation, inspection or testing, always use protective safety tools such as a cover for the device. Otherwise you may sustain injury caused by the device catching fire or exploding.

Make sure that all metal casings in your design are grounded to earth.
Even in modules where a device's electrodes and metal casing are insulated, capacitance in the module may cause the electrostatic potential in the casing to rise.
Dielectric breakdown may cause a high voltage to be applied to the casing, causing electric shock and injury to anyone touching it.

When designing the heat radiation and safety features of a system incorporating high-speed rectifiers, remember to take the device's forward and reverse losses into account.
The leakage current in these devices is greater than that in ordinary rectifiers; as a result, if a high-speed rectifier is used in an extreme environment (e.g. at high temperature or high voltage), its reverse loss may increase, causing thermal runaway to occur. This may in turn cause the device to explode and scatter shrapnel, resulting in injury to the user.

A design should ensure that, except when the main circuit of the device is active, reverse bias is applied to the device gate while electricity is conducted to control circuits, so that the main circuit will become inactive.
Malfunction of the device may cause serious accidents or injuries.

⚠CAUTION

When conducting any kind of evaluation, inspection or testing, either wear protective gloves or wait until the device has cooled properly before handling it.
Devices become hot when they are operated. Even after the power has been turned off, the device will retain residual heat which may cause a burn to anyone touching it.

2.2.3 Bipolar ICs (for use in automobiles)

⚠CAUTION

If your design includes an inductive load such as a motor coil, incorporate diodes or similar devices into the design to prevent negative current from flowing in.
The load current generated by powering the device on and off may cause it to function erratically or to break down, which could in turn cause injury.

Ensure that the power supply to any device which incorporates protective functions is stable.
If the power supply is unstable, the device may operate erratically, preventing the protective functions from working correctly. If protective functions fail, the device may break down causing injury to the user.

3. General Safety Precautions and Usage Considerations

This section is designed to help you gain a better understanding of semiconductor devices, so as to ensure the safety, quality and reliability of the devices which you incorporate into your designs.

3.1 From Incoming to Shipping

3.1.1 Electrostatic discharge (ESD)



When handling individual devices (which are not yet mounted on a printed circuit board), be sure that the environment is protected against electrostatic electricity. Operators should wear anti-static clothing, and containers and other objects which come into direct contact with devices should be made of anti-static materials and should be grounded to earth via an 0.5- to 1.0-MΩ protective resistor.

Please follow the precautions described below; this is particularly important for devices which are marked "Be careful of static.".

(1) Work environment

- When humidity in the working environment decreases, the human body and other insulators can easily become charged with static electricity due to friction. Maintain the recommended humidity of 40% to 60% in the work environment, while also taking into account the fact that moisture-proof-packed products may absorb moisture after unpacking.

- Be sure that all equipment, jigs and tools in the working area are grounded to earth.

- Place a conductive mat over the floor of the work area, or take other appropriate measures, so that the floor surface is protected against static electricity and is grounded to earth. The surface resistivity should be 10^4 to 10^8 Ω/sq and the resistance between surface and ground, 7.5×10^5 to 10^8 Ω

- Cover the workbench surface also with a conductive mat (with a surface resistivity of 10^4 to 10^8 Ω/sq, for a resistance between surface and ground of 7.5×10^5 to 10^8 Ω) . The purpose of this is to disperse static electricity on the surface (through resistive components) and ground it to earth. Workbench surfaces must not be constructed of low-resistance metallic materials that allow rapid static discharge when a charged device touches them directly.

- Pay attention to the following points when using automatic equipment in your workplace:

  (a) When picking up ICs with a vacuum unit, use a conductive rubber fitting on the end of the pick-up wand to protect against electrostatic charge.

  (b) Minimize friction on IC package surfaces. If some rubbing is unavoidable due to the device's mechanical structure, minimize the friction plane or use material with a small friction coefficient and low electrical resistance. Also, consider the use of an ionizer.

  (c) In sections which come into contact with device lead terminals, use a material which dissipates static electricity.

  (d) Ensure that no statically charged bodies (such as work clothes or the human body) touch the devices.

(e) Make sure that sections of the tape carrier which come into contact with installation devices or other electrical machinery are made of a low-resistance material.

(f) Make sure that jigs and tools used in the assembly process do not touch devices.

(g) In processes in which packages may retain an electrostatic charge, use an ionizer to neutralize the ions.

- Make sure that CRT displays in the working area are protected against static charge, for example by a VDT filter. As much as possible, avoid turning displays on and off. Doing so can cause electrostatic induction in devices.

- Keep track of charged potential in the working area by taking periodic measurements.

- Ensure that work chairs are protected by an anti-static textile cover and are grounded to the floor surface by a grounding chain. (Suggested resistance between the seat surface and grounding chain is 7.5×10^5 to $10^{12}\Omega$.)

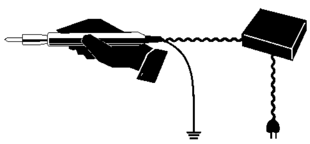
- Install anti-static mats on storage shelf surfaces. (Suggested surface resistivity is 10^4 to 10^8 Ω/sq; suggested resistance between surface and ground is 7.5×10^5 to 10^8 Ω.)

- For transport and temporary storage of devices, use containers (boxes, jigs or bags) that are made of anti-static materials or materials which dissipate electrostatic charge.

- Make sure that cart surfaces which come into contact with device packaging are made of materials which will conduct static electricity, and verify that they are grounded to the floor surface via a grounding chain.

- In any location where the level of static electricity is to be closely controlled, the ground resistance level should be Class 3 or above. Use different ground wires for all items of equipment which may come into physical contact with devices.

(2) Operating environment



- Operators must wear anti-static clothing and conductive shoes (or a leg or heel strap).

- Operators must wear a wrist strap grounded to earth via a resistor of about 1 MΩ.

- Soldering irons must be grounded from iron tip to earth, and must be used only at low voltages (6 V to 24 V).

- If the tweezers you use are likely to touch the device terminals, use anti-static tweezers and in particular avoid metallic tweezers. If a charged device touches a low-resistance tool, rapid discharge can occur. When using vacuum tweezers, attach a conductive chucking pat to the tip, and connect it to a dedicated ground used especially for anti-static purposes (suggested resistance value: 10^4 to 10^8 Ω).

- Do not place devices or their containers near sources of strong electrical fields (such as above a CRT).

- When storing printed circuit boards which have devices mounted on them, use a board container or bag that is protected against static charge. To avoid the occurrence of static charge or discharge due to friction, keep the boards separate from one other and do not stack them directly on top of one another.

- Ensure, if possible, that any articles (such as clipboards) which are brought to any location where the level of static electricity must be closely controlled are constructed of anti-static materials.

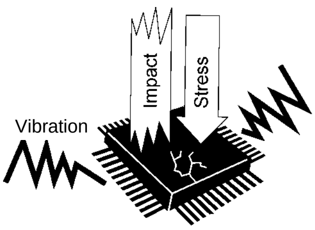
- In cases where the human body comes into direct contact with a device, be sure to wear anti-static finger covers or gloves (suggested resistance value: 10^8 Ω or less).

- Equipment safety covers installed near devices should have resistance ratings of 10^9 Ω or less.

- If a wrist strap cannot be used for some reason, and there is a possibility of imparting friction to devices, use an ionizer.

- The transport film used in TCP products is manufactured from materials in which static charges tend to build up. When using these products, install an ionizer to prevent the film from being charged with static electricity. Also, ensure that no static electricity will be applied to the product's copper foils by taking measures to prevent static occuring in the peripheral equipment.

3.1.2 Vibration, impact and stress



Handle devices and packaging materials with care. To avoid damage to devices, do not toss or drop packages. Ensure that devices are not subjected to mechanical vibration or shock during transportation. Ceramic package devices and devices in canister-type packages which have empty space inside them are subject to damage from vibration and shock because the bonding wires are secured only at their ends.

Plastic molded devices, on the other hand, have a relatively high level of resistance to vibration and mechanical shock because their bonding wires are enveloped and fixed in resin. However, when any device or package type is installed in target equipment, it is to some extent susceptible to wiring disconnections and other damage from vibration, shock and stressed solder junctions. Therefore when devices are incorporated into the design of equipment which will be subject to vibration, the structural design of the equipment must be thought out carefully.

If a device is subjected to especially strong vibration, mechanical shock or stress, the package or the chip itself may crack. In products such as CCDs which incorporate window glass, this could cause surface flaws in the glass or cause the connection between the glass and the ceramic to separate.

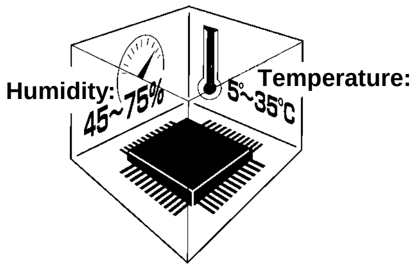
Furthermore, it is known that stress applied to a semiconductor device through the package changes the resistance characteristics of the chip because of piezoelectric effects. In analog circuit design attention must be paid to the problem of package stress as well as to the dangers of vibration and shock as described above.

3.2 Storage

3.2.1 General storage

- Avoid storage locations where devices will be exposed to moisture or direct sunlight.

- Follow the instructions printed on the device cartons regarding transportation and storage.



- The storage area temperature should be kept within a temperature range of 5°C to 35°C, and relative humidity should be maintained at between 45% and 75%.

- Do not store devices in the presence of harmful (especially corrosive) gases, or in dusty conditions.

- Use storage areas where there is minimal temperature fluctuation. Rapid temperature changes can cause moisture to form on stored devices, resulting in lead oxidation or corrosion. As a result, the solderability of the leads will be degraded.

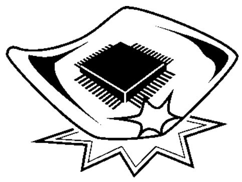
- When repacking devices, use anti-static containers.

- Do not allow external forces or loads to be applied to devices while they are in storage.

- If devices have been stored for more than two years, their electrical characteristics should be tested and their leads should be tested for ease of soldering before they are used.

3.2.2 Moisture-proof packing

Moisture-proof packing should be handled with care. The handling procedure specified for each packing type should be followed scrupulously. If the proper procedures are not followed, the quality and reliability of devices may be degraded. This section describes general precautions for handling moisture-proof packing. Since the details may differ from device to device, refer also to the relevant individual datasheets or databook.



(1) General precautions
Follow the instructions printed on the device cartons regarding transportation and storage.

- Do not drop or toss device packing. The laminated aluminum material in it can be rendered ineffective by rough handling.

- The storage area temperature should be kept within a temperature range of 5°C to 30°C, and relative humidity should be maintained at 90% (max). Use devices within 12 months of the date marked on the package seal.

- If the 12-month storage period has expired, or if the 30% humidity indicator shown in Figure 1 is pink when the packing is opened, it may be advisable, depending on the device and packing type, to back the devices at high temperature to remove any moisture. Please refer to the table below. After the pack has been opened, use the devices in a 5°C to 30°C. 60% RH environment and within the effective usage period listed on the moisture-proof package. If the effective usage period has expired, or if the packing has been stored in a high-humidity environment, back the devices at high temperature.

Packing	Moisture removal
Tray	If the packing bears the "Heatproof" marking or indicates the maximum temperature which it can withstand, bake at 125°C for 20 hours. (Some devices require a different procedure.)
Tube	Transfer devices to trays bearing the "Heatproof" marking or indicating the temperature which they can withstand, or to aluminum tubes before baking at 125°C for 20 hours.
Tape	Deviced packed on tape cannot be baked and must be used within the effective usage period after unpacking, as specified on the packing.

- When baking devices, protect the devices from static electricity.

- Moisture indicators can detect the approximate humidity level at a standard temperature of 25°C. 6-point indicators and 3-point indicators are currently in use, but eventually all indicators will be 3-point indicators.

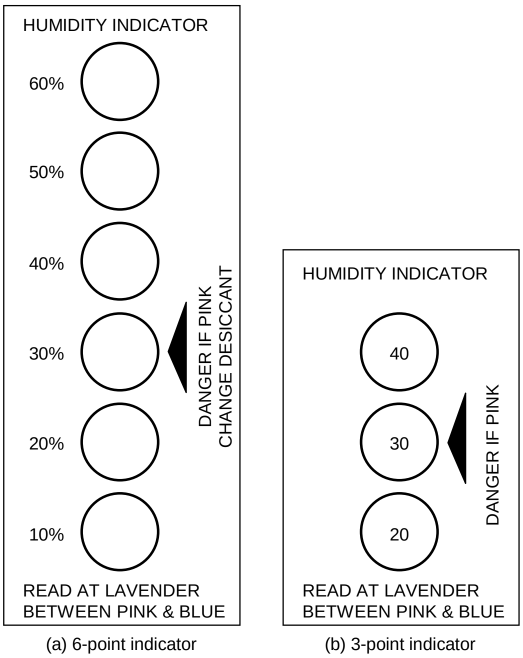


Figure 1 Humidity indicator

3.3 Design

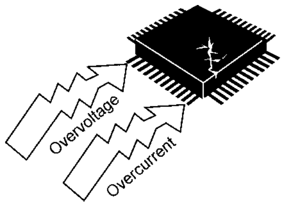
Care must be exercised in the design of electronic equipment to achieve the desired reliability. It is important not only to adhere to specifications concerning absolute maximum ratings and recommended operating conditions, it is also important to consider the overall environment in which equipment will be used, including factors such as the ambient temperature, transient noise and voltage and current surges, as well as mounting conditions which affect device reliability. This section describes some general precautions which you should observe when designing circuits and when mounting devices on printed circuit boards.

For more detailed information about each product family, refer to the relevant individual technical datasheets available from Toshiba.

3.3.1 Absolute maximum ratings

CAUTION Do not use devices under conditions in which their absolute maximum ratings (e.g. current, voltage, power dissipation or temperature) will be exceeded. A device may break down or its performance may be degraded, causing it to catch fire or explode resulting in injury to the user.

The absolute maximum ratings are rated values which must not be exceeded during operation, even for an instant. Although absolute maximum ratings differ from product to product, they essentially concern the voltage and current at each pin, the allowable power dissipation, and the junction and storage temperatures.



If the voltage or current on any pin exceeds the absolute maximum rating, the device's internal circuitry can become degraded. In the worst case, heat generated in internal circuitry can fuse wiring or cause the semiconductor chip to break down.

If storage or operating temperatures exceed rated values, the package seal can deteriorate or the wires can become disconnected due to the differences between the thermal expansion coefficients of the materials from which the device is constructed.

3.3.2 Recommended operating conditions

The recommended operating conditions for each device are those necessary to guarantee that the device will operate as specified in the datasheet.
If greater reliability is required, derate the device's absolute maximum ratings for voltage, current, power and temperature before using it.

3.3.3 Derating

When incorporating a device into your design, reduce its rated absolute maximum voltage, current, power dissipation and operating temperature in order to ensure high reliability.
Since derating differs from application to application, refer to the technical datasheets available for the various devices used in your design.

3.3.4 Unused pins

If unused pins are left open, some devices can exhibit input instability problems, resulting in malfunctions such as abrupt increase in current flow. Similarly, if the unused output pins on a device are connected to the power supply pin, the ground pin or to other output pins, the IC may malfunction or break down.

Since the details regarding the handling of unused pins differ from device to device and from pin to pin, please follow the instructions given in the relevant individual datasheets or databook.

CMOS logic IC inputs, for example, have extremely high impedance. If an input pin is left open, it can easily pick up extraneous noise and become unstable. In this case, if the input voltage level reaches an intermediate level, it is possible that both the P-channel and N-channel transistors will be turned on, allowing unwanted supply current to flow. Therefore, ensure that the unused input pins of a device are connected to the power supply (Vcc) pin or ground (GND) pin of the same device. For details of what to do with the pins of heat sinks, refer to the relevant technical datasheet and databook.

3.3.5 Latch-up

Latch-up is an abnormal condition inherent in CMOS devices, in which Vcc gets shorted to ground. This happens when a parasitic PN-PN junction (thyristor structure) internal to the CMOS chip is turned on, causing a large current of the order of several hundred mA or more to flow between Vcc and GND, eventually causing the device to break down.

Latch-up occurs when the input or output voltage exceeds the rated value, causing a large current to flow in the internal chip, or when the voltage on the Vcc (Vdd) pin exceeds its rated value, forcing the internal chip into a breakdown condition. Once the chip falls into the latch-up state, even though the excess voltage may have been applied only for an instant, the large current continues to flow between Vcc (Vdd) and GND (Vss). This causes the device to heat up and, in extreme cases, to emit gas fumes as well. To avoid this problem, observe the following precautions:

(1) Do not allow voltage levels on the input and output pins either to rise above Vcc (Vdd) or to fall below GND (Vss). Also, follow any prescribed power-on sequence, so that power is applied gradually or in steps rather than abruptly.

(2) Do not allow any abnormal noise signals to be applied to the device.

(3) Set the voltage levels of unused input pins to Vcc (Vdd) or GND (Vss).

(4) Do not connect output pins to one another.

3.3.6 Input/Output protection

Wired-AND configurations, in which outputs are connected together, cannot be used, since this short-circuits the outputs. Outputs should, of course, never be connected to Vcc (Vdd) or GND (Vss).

Furthermore, ICs with tri-state outputs can undergo performance degradation if a shorted output current is allowed to flow for an extended period of time. Therefore, when designing circuits, make sure that tri-state outputs will not be enabled simultaneously.

3.3.7 Load capacitance

Some devices display increased delay times if the load capacitance is large. Also, large charging and discharging currents will flow in the device, causing noise. Furthermore, since outputs are shorted for a relatively long time, wiring can become fused.

Consult the technical information for the device being used to determine the recommended load capacitance.

3.3.8 Thermal design

The failure rate of semiconductor devices is greatly increased as operating temperatures increase. As shown in Figure 2, the internal thermal stress on a device is the sum of the ambient temperature and the temperature rise due to power dissipation in the device. Therefore, to achieve optimum reliability, observe the following precautions concerning thermal design:

(1) Keep the ambient temperature (Ta) as low as possible.

(2) If the device's dynamic power dissipation is relatively large, select the most appropriate circuit board material, and consider the use of heat sinks or of forced air cooling. Such measures will help lower the thermal resistance of the package.

(3) Derate the device's absolute maximum ratings to minimize thermal stress from power dissipation.

$\theta ja = \theta jc + \theta ca$

$\theta ja = (Tj - Ta) / P$

$\theta jc = (Tj - Tc) / P$

$\theta ca = (Tc - Ta) / P$

in which θja = thermal resistance between junction and surrounding air (°C/W)

θjc = thermal resistance between junction and package surface, or internal thermal resistance (°C/W)

θca = thermal resistance between package surface and surrounding air, or external thermal resistance (°C/W)

Tj = junction temperature or chip temperature (°C)

Tc = package surface temperature or case temperature (°C)

Ta = ambient temperature (°C)

P = power dissipation (W)

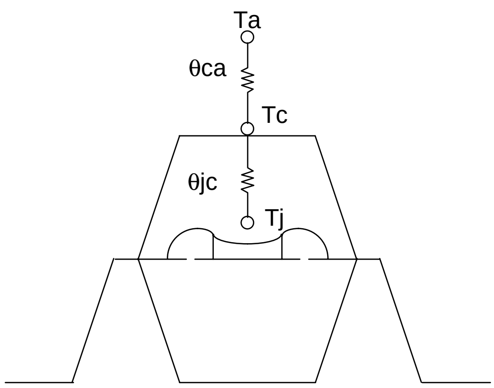


Figure 2 Thermal resistance of package

3.3.9 Interfacing

When connecting inputs and outputs between devices, make sure input voltage (V_{IL}/V_{IH}) and output voltage (V_{OL}/V_{OH}) levels are matched. Otherwise, the devices may malfunction. When connecting devices operating at different supply voltages, such as in a dual-power-supply system, be aware that erroneous power-on and power-off sequences can result in device breakdown. For details of how to interface particular devices, consult the relevant technical datasheets and databooks. If you have any questions or doubts about interfacing, contact your nearest Toshiba office or distributor.

3.3.10 Decoupling

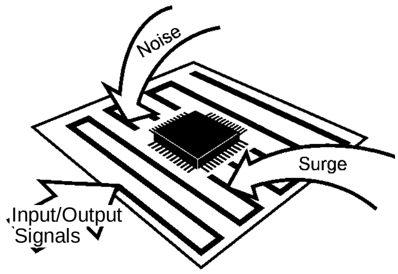
Spike currents generated during switching can cause Vcc (Vdd) and GND (Vss) voltage levels to fluctuate, causing ringing in the output waveform or a delay in response speed. (The power supply and GND wiring impedance is normally 50 Ω to 100 Ω.) For this reason, the impedance of power supply lines with respect to high frequencies must be kept low. This can be accomplished by using thick and short wiring for the Vcc (Vdd) and GND (Vss) lines and by installing decoupling capacitors (of approximately 0.01 µF to 1 µF capacitance) as high-frequency filters between Vcc (Vdd) and GND (Vss) at strategic locations on the printed circuit board.

For low-frequency filtering, it is a good idea to install a 10- to 100-µF capacitor on the printed circuit board (one capacitor will suffice). If the capacitance is excessively large, however, (e.g. several thousand µF) latch-up can be a problem. Be sure to choose an appropriate capacitance value.

An important point about wiring is that, in the case of high-speed logic ICs, noise is caused mainly by reflection and crosstalk, or by the power supply impedance. Reflections cause increased signal delay, ringing, overshoot and undershoot, thereby reducing the device's safety margins with respect to noise. To prevent reflections, reduce the wiring length by increasing the device mounting density so as to lower the inductance (L) and capacitance (C) in the wiring. Extreme care must be taken, however, when taking this corrective measure, since it tends to cause crosstalk between the wires. In practice, there must be a trade-off between these two factors.

3.3.11 External noise

Printed circuit boards with long I/O or signal pattern lines are vulnerable to induced noise or surges from outside sources. Consequently, malfunctions or breakdowns can result from overcurrent or overvoltage, depending on the types of device used. To protect against noise, lower the impedance of the pattern line or insert a noise-canceling circuit. Protective measures must also be taken against surges.



For details of the appropriate protective measures for a particular device, consult the relevant databook.

3.3.12 Electromagnetic interference

Widespread use of electrical and electronic equipment in recent years has brought with it radio and TV reception problems due to electromagnetic interference. To use the radio spectrum effectively and to maintain radio communications quality, each country has formulated regulations limiting the amount of electromagnetic interference which can be generated by individual products.

Electromagnetic interference includes conduction noise propagated through power supply and telephone lines, and noise from direct electromagnetic waves radiated by equipment. Different measurement methods and corrective measures are used to assess and counteract each specific type of noise.

Difficulties in controlling electromagnetic interference derive from the fact that there is no method available which allows designers to calculate, at the design stage, the strength of the electromagnetic waves which will emanate from each component in a piece of equipment. For this reason, it is only after the prototype equipment has been completed that the designer can take measurements using a dedicated instrument to determine the strength of electromagnetic interference waves. Yet it is possible during system design to incorporate some measures for the prevention of electromagnetic interference, which can facilitate taking corrective measures once the design has been completed. These include installing shields and noise filters, and increasing

the thickness of the power supply wiring patterns on the printed circuit board. One effective method, for example, is to devise several shielding options during design, and then select the most suitable shielding method based on the results of measurements taken after the prototype has been completed.

3.3.13 Peripheral circuits

In most cases semiconductor devices are used with peripheral circuits and components. The input and output signal voltages and currents in these circuits must be chosen to match the semiconductor device's specifications. The following factors must be taken into account.

(1) Inappropriate voltages or currents applied to a device's input pins may cause it to operate erratically. Some devices contain pull-up or pull-down resistors. When designing your system, remember to take the effect of this on the voltage and current levels into account.

(2) The output pins on a device have a predetermined external circuit drive capability. If this drive capability is greater than that required, either incorporate a compensating circuit into your design or carefully select suitable components for use in external circuits.

3.3.14 Safety standards

Each country has safety standards which must be observed. These safety standards include requirements for quality assurance systems and design of device insulation. Such requirements must be fully taken into account to ensure that your design conforms to the applicable safety standards.

3.3.15 Other precautions

(1) When designing a system, be sure to incorporate fail-safe and other appropriate measures according to the intended purpose of your system. Also, be sure to debug your system under actual board-mounted conditions.

(2) If a plastic-package device is placed in a strong electric field, surface leakage may occur due to the charge-up phenomenon, resulting in device malfunction. In such cases take appropriate measures to prevent this problem, for example by protecting the package surface with a conductive shield.

(3) With some microcomputers and MOS memory devices, caution is required when powering on or resetting the device. To ensure that your design does not violate device specifications, consult the relevant databook for each constituent device.

(4) Ensure that no conductive material or object (such as a metal pin) can drop onto and short the leads of a device mounted on a printed circuit board.

3.4 Inspection, Testing and Evaluation

3.4.1 Grounding

⚠CAUTION Ground all measuring instruments, jigs, tools and soldering irons to earth. Electrical leakage may cause a device to break down or may result in electric shock.

3.4.2 Inspection Sequence

⚠CAUTION

① Do not insert devices in the wrong orientation. Make sure that the positive and negative electrodes of the power supply are correctly connected. Otherwise, the rated maximum current or maximum power dissipation may be exceeded and the device may break down or undergo performance degradation, causing it to catch fire or explode, resulting in injury to the user.

② When conducting any kind of evaluation, inspection or testing using AC power with a peak voltage of 42.4 V or DC power exceeding 60 V, be sure to connect the electrodes or probes of the testing equipment to the device under test before powering it on. Connecting the electrodes or probes of testing equipment to a device while it is powered on may result in electric shock, causing injury.

(1) Apply voltage to the test jig only after inserting the device securely into it. When applying or removing power, observe the relevant precautions, if any.

(2) Make sure that the voltage applied to the device is off before removing the device from the test jig. Otherwise, the device may undergo performance degradation or be destroyed.

(3) Make sure that no surge voltages from the measuring equipment are applied to the device.

(4) The chips housed in tape carrier packages (TCPs) are bare chips and are therefore exposed. During inspection take care not to crack the chip or cause any flaws in it.
Electrical contact may also cause a chip to become faulty. Therefore make sure that nothing comes into electrical contact with the chip.

3.5 Mounting

There are essentially two main types of semiconductor device package: lead insertion and surface mount. During mounting on printed circuit boards, devices can become contaminated by flux or damaged by thermal stress from the soldering process. With surface-mount devices in particular, the most significant problem is thermal stress from solder reflow, when the entire package is subjected to heat. This section describes a recommended temperature profile for each mounting method, as well as general precautions which you should take when mounting devices on printed circuit boards. Note, however, that even for devices with the same package type, the appropriate mounting method varies according to the size of the chip and the size and shape of the lead frame. Therefore, please consult the relevant technical datasheet and databook.

3.5.1 Lead forming

⚠CAUTION

① Always wear protective glasses when cutting the leads of a device with clippers or a similar tool. If you do not, small bits of metal flying off the cut ends may damage your eyes.

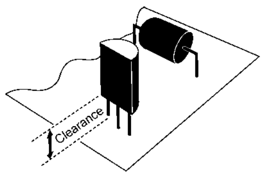
② Do not touch the tips of device leads. Because some types of device have leads with pointed tips, you may prick your finger.

Semiconductor devices must undergo a process in which the leads are cut and formed before the devices can be mounted on a printed circuit board. If undue stress is applied to the interior of a device during this process, mechanical breakdown or performance degradation can result. This is attributable primarily to differences between the stress on the device's external leads and the stress on the internal leads. If the relative difference is great enough, the device's internal leads, adhesive properties or sealant can be damaged. Observe these precautions during the lead-forming process (this does not apply to surface-mount devices):

(1) Lead insertion hole intervals on the printed circuit board should match the lead pitch of the device precisely.

(2) If lead insertion hole intervals on the printed circuit board do not precisely match the lead pitch of the device, do not attempt to forcibly insert devices by pressing on them or by pulling on their leads.

(3) For the minimum clearance specification between a device and a printed circuit board, refer to the relevant device's datasheet and databook. If necessary, achieve the required clearance by forming the device's leads appropriately. Do not use the spacers which are used to raise devices above the surface of the printed circuit board during soldering to achieve clearance. These spacers normally continue to expand due to heat, even after the solder has begun to solidify; this applies severe stress to the device.



(4) Observe the following precautions when forming the leads of a device prior to mounting.

- Use a tool or jig to secure the lead at its base (where the lead meets the device package) while bending so as to avoid mechanical stress to the device. Also avoid bending or stretching device leads repeatedly.

- Be careful not to damage the lead during lead forming.

- Follow any other precautions described in the individual datasheets and databooks for each device and package type.

3.5.2 Socket mounting

(1) When socket mounting devices on a printed circuit board, use sockets which match the inserted device's package.

(2) Use sockets whose contacts have the appropriate contact pressure. If the contact pressure is insufficient, the socket may not make a perfect contact when the device is repeatedly inserted and removed; if the pressure is excessively high, the device leads may be bent or damaged when they are inserted into or removed from the socket.

(3) When soldering sockets to the printed circuit board, use sockets whose construction prevents flux from penetrating into the contacts or which allows flux to be completely cleaned off.

(4) Make sure the coating agent applied to the printed circuit board for moisture-proofing purposes does not stick to the socket contacts.

(5) If the device leads are severely bent by a socket as it is inserted or removed and you wish to repair the leads so as to continue using the device, make sure that this lead correction is only performed once. Do not use devices whose leads have been corrected more than once.

(6) If the printed circuit board with the devices mounted on it will be subjected to vibration from external sources, use sockets which have a strong contact pressure so as to prevent the sockets and devices from vibrating relative to one another.

3.5.3 Soldering temperature profile

The soldering temperature and heating time vary from device to device. Therefore, when specifying the mounting conditions, refer to the individual datasheets and databooks for the devices used.

(1) Using a soldering iron

Complete soldering within ten seconds for lead temperatures of up to 260°C, or within three seconds for lead temperatures of up to 350°C.

(2) Using medium infrared ray reflow

- Heating top and bottom with long or medium infrared rays is recommended (see Figure 3).

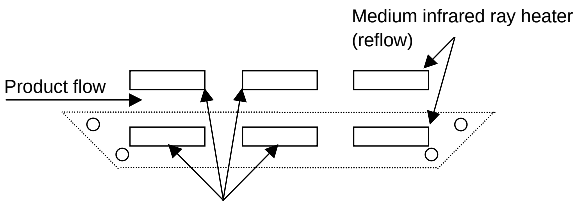


Figure 3 Heating top and bottom with long or medium infrared rays

- Complete the infrared ray reflow process within 30 seconds at a package surface temperature of between 210°C and 240°C.
- Refer to Figure 4 for an example of a good temperature profile for infrared or hot air reflow.

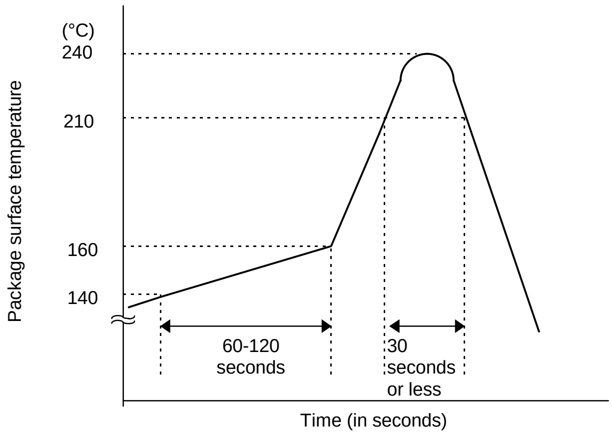


Figure 4 Sample temperature profile for infrared or hot air reflow

(3) Using hot air reflow

- Complete hot air reflow within 30 seconds at a package surface temperature of between 210°C and 240°C.
- For an example of a recommended temperature profile, refer to Figure 4 above.

(4) Using solder flow

- Apply preheating for 60 to 120 seconds at a temperature of 150°C.
- For lead insertion-type packages, complete solder flow within 10 seconds with the temperature at the stopper (or, if there is no stopper, at a location more than 1.5 mm from the body) which does not exceed 260°C.

- For surface-mount packages, complete soldering within 5 seconds at a temperature of 250°C or less in order to prevent thermal stress in the device.

- Figure 5 shows an example of a recommended temperature profile for surface-mount packages using solder flow.

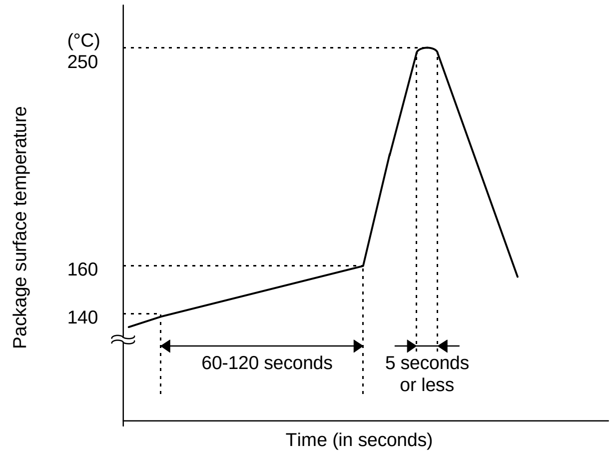


Figure 5 Sample temperature profile for solder flow

3.5.4 Flux cleaning and ultrasonic cleaning

(1) When cleaning circuit boards to remove flux, make sure that no residual reactive ions such as Na or Cl remain. Note that organic solvents react with water to generate hydrogen chloride and other corrosive gases which can degrade device performance.

(2) Washing devices with water will not cause any problems. However, make sure that no reactive ions such as sodium and chlorine are left as a residue. Also, be sure to dry devices sufficiently after washing.

(3) Do not rub device markings with a brush or with your hand during cleaning or while the devices are still wet from the cleaning agent. Doing so can rub off the markings.

(4) The dip cleaning, shower cleaning and steam cleaning processes all involve the chemical action of a solvent. Use only recommended solvents for these cleaning methods. When immersing devices in a solvent or steam bath, make sure that the temperature of the liquid is 50°C or below, and that the circuit board is removed from the bath within one minute.

(5) Ultrasonic cleaning should not be used with hermetically-sealed ceramic packages such as a leadless chip carrier (LCC), pin grid array (PGA) or charge-coupled device (CCD), because the bonding wires can become disconnected due to resonance during the cleaning process. Even if a device package allows ultrasonic cleaning, limit the duration of ultrasonic cleaning to as short a time as possible, since long hours of ultrasonic cleaning degrade the adhesion between the mold resin and the frame material. The following ultrasonic cleaning conditions are recommended:

    Frequency: 27 kHz ~ 29 kHz

    Ultrasonic output power: 300 W or less (0.25 W/cm^2 or less)

    Cleaning time: 30 seconds or less

    Suspend the circuit board in the solvent bath during ultrasonic cleaning in such a way that the ultrasonic vibrator does not come into direct contact with the circuit board or the device.

3.5.5 No cleaning

If analog devices or high-speed devices are used without being cleaned, flux residues may cause minute amounts of leakage between pins. Similarly, dew condensation, which occurs in environments containing residual chlorine when power to the device is on, may cause between-lead leakage or migration. Therefore, Toshiba recommends that these devices be cleaned. However, if the flux used contains only a small amount of halogen (0.05W% or less), the devices may be used without cleaning without any problems.

3.5.6 Mounting tape carrier packages (TCPs)

(1) When tape carrier packages (TCPs) are mounted, measures must be taken to prevent electrostatic breakdown of the devices.

(2) If devices are being picked up from tape, or outer lead bonding (OLB) mounting is being carried out, consult the manufacturer of the insertion machine which is being used, in order to establish the optimum mounting conditions in advance and to avoid any possible hazards.

(3) The base film, which is made of polyimide, is hard and thin. Be careful not to cut or scratch your hands or any objects while handling the tape.

(4) When punching tape, try not to scatter broken pieces of tape too much.

(5) Treat the extra film, reels and spacers left after punching as industrial waste, taking care not to destroy or pollute the environment.

(6) Chips housed in tape carrier packages (TCPs) are bare chips and therefore have their reverse side exposed. To ensure that the chip will not be cracked during mounting, ensure that no mechanical shock is applied to the reverse side of the chip. Electrical contact may also cause a chip to fail. Therefore, when mounting devices, make sure that nothing comes into electrical contact with the reverse side of the chip.
If your design requires connecting the reverse side of the chip to the circuit board, please consult Toshiba or a Toshiba distributor beforehand.

3.5.7 Mounting chips

Devices delivered in chip form tend to degrade or break under external forces much more easily than plastic-packaged devices. Therefore, caution is required when handling this type of device.

(1) Mount devices in a properly prepared environment so that chip surfaces will not be exposed to polluted ambient air or other polluted substances.

(2) When handling chips, be careful not to expose them to static electricity.
In particular, measures must be taken to prevent static damage during the mounting of chips. With this in mind, Toshiba recommend mounting all peripheral parts first and then mounting chips last (after all other components have been mounted).

(3) Make sure that PCBs (or any other kind of circuit board) on which chips are being mounted do not have any chemical residues on them (such as the chemicals which were used for etching the PCBs).

(4) When mounting chips on a board, use the method of assembly that is most suitable for maintaining the appropriate electrical, thermal and mechanical properties of the semiconductor devices used.

* For details of devices in chip form, refer to the relevant device's individual datasheets.

3.5.8 Circuit board coating

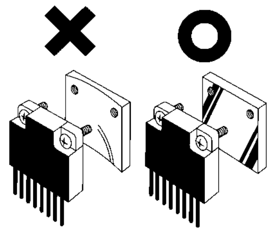
When devices are to be used in equipment requiring a high degree of reliability or in extreme environments (where moisture, corrosive gas or dust is present), circuit boards may be coated for protection. However, before doing so, you must carefully consider the possible stress and contamination effects that may result and then choose the coating resin which results in the minimum level of stress to the device.

3.5.9 Heat sinks

(1) When attaching a heat sink to a device, be careful not to apply excessive force to the device in the process.

(2) When attaching a device to a heat sink by fixing it at two or more locations, evenly tighten all the screws in stages (i.e. do not fully tighten one screw while the rest are still only loosely tightened). Finally, fully tighten all the screws up to the specified torque.

(3) Drill holes for screws in the heat sink exactly as specified. Smooth the surface by removing burrs and protrusions or indentations which might interfere with the installation of any part of the device.



(4) A coating of silicone compound can be applied between the heat sink and the device to improve heat conductivity. Be sure to apply the coating thinly and evenly; do not use too much. Also, be sure to use a non-volatile compound, as volatile compounds can crack after a time, causing the heat radiation properties of the heat sink to deteriorate.

(5) If the device is housed in a plastic package, use caution when selecting the type of silicone compound to be applied between the heat sink and the device. With some types, the base oil separates and penetrates the plastic package, significantly reducing the useful life of the device.
Two recommended silicone compounds in which base oil separation is not a problem are YG6260 from Toshiba Silicone.

(6) Heat-sink-equipped devices can become very hot during operation. Do not touch them, or you may sustain a burn.

3.5.10 Tightening torque

(1) Make sure the screws are tightened with fastening torques not exceeding the torque values stipulated in individual datasheets and databooks for the devices used.

(2) Do not allow a power screwdriver (electrical or air-driven) to touch devices.

3.5.11 Repeated device mounting and usage

Do not remount or re-use devices which fall into the categories listed below; these devices may cause significant problems relating to performance and reliability.

(1) Devices which have been removed from the board after soldering

(2) Devices which have been inserted in the wrong orientation or which have had reverse current applied

(3) Devices which have undergone lead forming more than once

3.6 Protecting Devices in the Field

3.6.1 Temperature

Semiconductor devices are generally more sensitive to temperature than are other electronic components. The various electrical characteristics of a semiconductor device are dependent on the ambient temperature at which the device is used. It is therefore necessary to understand the temperature characteristics of a device and to incorporate device derating into circuit design. Note also that if a device is used above its maximum temperature rating, device deterioration is more rapid and it will reach the end of its usable life sooner than expected.

3.6.2 Humidity

Resin-molded devices are sometimes improperly sealed. When these devices are used for an extended period of time in a high-humidity environment, moisture can penetrate into the device and cause chip degradation or malfunction. Furthermore, when devices are mounted on a regular printed circuit board, the impedance between wiring components can decrease under high-humidity conditions. In systems which require a high signal-source impedance, circuit board leakage or leakage between device lead pins can cause malfunctions. The application of a moisture-proof treatment to the device surface should be considered in this case. On the other hand, operation under low-humidity conditions can damage a device due to the occurrence of electrostatic discharge. Unless damp-proofing measures have been specifically taken, use devices only in environments with appropriate ambient moisture levels (i.e. within a relative humidity range of 40% to 60%).

3.6.3 Corrosive gases

Corrosive gases can cause chemical reactions in devices, degrading device characteristics.
For example, sulphur-bearing corrosive gases emanating from rubber placed near a device (accompanied by condensation under high-humidity conditions) can corrode a device's leads. The resulting chemical reaction between leads forms foreign particles which can cause electrical leakage.

3.6.4 Radioactive and cosmic rays

Most industrial and consumer semiconductor devices are not designed with protection against radioactive and cosmic rays. Devices used in aerospace equipment or in radioactive environments must therefore be shielded.

3.6.5 Strong electrical and magnetic fields

Devices exposed to strong magnetic fields can undergo a polarization phenomenon in their plastic material, or within the chip, which gives rise to abnormal symptoms such as impedance changes or increased leakage current. Failures have been reported in LSIs mounted near malfunctioning deflection yokes in TV sets. In such cases the device's installation location must be changed or the device must be shielded against the electrical or magnetic field. Shielding against magnetism is especially necessary for devices used in an alternating magnetic field because of the electromotive forces generated in this type of environment.

3.6.6 Interference from light (ultraviolet rays, sunlight, fluorescent lamps and incandescent lamps)

Light striking a semiconductor device generates electromotive force due to photoelectric effects. In some cases the device can malfunction. This is especially true for devices in which the internal chip is exposed. When designing circuits, make sure that devices are protected against incident light from external sources. This problem is not limited to optical semiconductors and EPROMs. All types of device can be affected by light.

3.6.7 Dust and oil

Just like corrosive gases, dust and oil can cause chemical reactions in devices, which will adversely affect a device's electrical characteristics. To avoid this problem, do not use devices in dusty or oily environments. This is especially important for optical devices because dust and oil can affect a device's optical characteristics as well as its physical integrity and the electrical performance factors mentioned above.

3.6.8 Fire

Semiconductor devices are combustible; they can emit smoke and catch fire if heated sufficiently. When this happens, some devices may generate poisonous gases. Devices should therefore never be used in close proximity to an open flame or a heat-generating body, or near flammable or combustible materials.

3.7 Disposal of devices and packing materials

When discarding unused devices and packing materials, follow all procedures specified by local regulations in order to protect the environment against contamination.

4. Precautions and Usage Considerations

This section describes matters specific to each product group which need to be taken into consideration when using devices. If the same item is described in Sections 3 and 4, the description in Section 4 takes precedence.

4.1 Microcontrollers

4.1.1 Design

(1) Using resonators which are not specifically recommended for use

Resonators recommended for use with Toshiba products in microcontroller oscillator applications are listed in Toshiba databooks along with information about oscillation conditions. If you use a resonator not included in this list, please consult Toshiba or the resonator manufacturer concerning the suitability of the device for your application.

(2) Undefined functions

In some microcontrollers certain instruction code values do not constitute valid processor instructions. Also, it is possible that the values of bits in registers will become undefined. Take care in your applications not to use invalid instructions or to let register bit values become undefined.

1. Introduction

This user's manual describes the C790 superscalar microprocessor for the system designer, paying special attention to the software interface and the bus interface.

The C790 is a superscalar integrated implementation of the subset of the 64-bit MIPS IV Instruction Set Architecture. It also implements a large extension to this instruction set specially tailored for multimedia applications. It contains a CPU, a floating point execution unit (Coprocessor 1), primary instruction and data caches.

Two instructions can be decoded each cycle. These instructions are issued in-order and are always completed in-order[1]. Data cache misses are non-blocking. A single outstanding cache miss does not stall the pipeline, so that load misses or uncached loads are retired out-of-order. Multiply, Multiply-Accumulate, Divide, Prefetch, and Coprocessor 1 instructions are also retired out-of-order.

[1] However, some instructions are retired out-of-order.

1.1 Features

The C790 core has the following features:

- 2-way superscalar pipeline
- 128-bit (two 64-bit) data path and 128-bit system bus
- Instruction set architecture
    - 64-bit MIPS III instruction set implementation (except LL, SC, LLD and SCD)
    - Selected MIPS IV instruction set implementation (Prefetch and Move conditional instructions)
    - Three-operand Multiply and Multiply-Accumulate instructions
    - 128-bit (Quadword) load/store instructions
    - 128-bit multimedia instructions which configure the 128-bit data path as two 64-bit, four 32-bit, eight 16-bit or sixteen 8-bit paths
    - Configurable Endianness
- Branch prediction with Branch History Table (BHT) and Branch Target Address Cache (BTAC)
- Large on-chip caches
    - Instruction cache: 32KB, 2-way set associative
    - Data cache: 32KB, 2-way set-associative (with write-back protocol)
    - Non-blocking load, hit under miss and early restart on first quadword
    - Data cache line locking
    - Prefetch functions
    - 64 Byte cache line
- Fast integer Multiply and Multiply-Accumulate operations
- Memory management unit
    - 48-entry (96 pages) fully associative translation look-aside buffer (TLB)
    - 32-bit physical address space and 32-bit virtual address space
- IEEE754-1985 compatible FPU (MIPS III ISA supported)
- Performance counters supported
- Debug support
    - Multi-stepping of instruction execution
    - Hardware breakpoint on instruction addresses
    - Hardware breakpoint on data address and data value
    - PC tracing capability
- 128-bit demultiplexed data bus and 32-bit address bus
    - Pipelined addresses
    - Bus error supported
    - Multiple masters supported

1.2 Related Documents

The following documents should be referenced:

[1] MIPS R4000 Microprocessor User’s Manual

[2] MIPS R10000 Microprocessor User’s Manual

[3] MIPS IV Instruction Set (Revision 3.2)

1.3 Revision History

Rev. 1.0: June 24th, 1999

Rev. 1.1: December 25th, 1999

Add IEEE754 compatible FPU feature (both single- and double-precision)

Rev. 1.2: March , 2000

Publish

Rev. 2.0: April , 2001

Fixed a lot of typo

1.4 Conventions Used in This Manual

The names of registers, fields, and instructions are *italicized* as in this example:

> The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and diagnostic states of the processor.

When a name is first introduced, it is shown in **bold type.**

Ranges are denoted by a colon as in the following example:

> The 4-bit *Coprocessor Usability (CU[3:0])* field controls the usability of four possible coprocessors.

Conventions used in instruction descriptions are defined at the beginning of Appendices A, B, C, and D.

1.5 Restrictions for Use of the C790 CPU Core

1. Revision History

Revision	Date	Contents
1.0	4/2/2001	FLX01-FLX06; Restrictions for User's Manual Rev.2.0

Items 1 through 6 in the description below are the restrictions that must be obeyed when using the C790 CPU core (User's Manual Rev.2.0).

Table 1-1. Restriction List

ID	Contents
FLX01	TLB exceptions masks bus errors.
FLX02	Bus errors are masked when Status.ERL==1 or Status.EXL = 1.
FLX03	AdEL occurs in index-type ICACHE or BTAC CACHE instructions.
FLX04	kuseg becomes an uncached area when an error exception (Status.ERL = 1) occurs.
FLX05	First two instructions in an exception handler are executed as NOP when a bus error occurs.
FLX06	Unexpected instruction-fetch bus-errors occur when executing a Crashme program.

2. Description

2.1 TLB exceptions mask bus errors (FLX01)

2.1.1 Phenomenon

There are cases in which TLB exceptions occurring immediately after a bus error mask the bus error and the bus error can not be detected.

2.1.2 Corrective measures

This is caused by bus error exceptions having a lower priority than TLB exceptions in instruction fetch and data access (refer to “5.5.1 Exception Priority”). Check the followings when programming a TLB exception handler.

1) Using the TLB exception handler, check for occurrence of any bus error exceptions before a page refill.

2) Using the TLB exception handler, check for occurrence of any bus error exceptions if a page that should be refilled is incorrect.

3) Using the TLB exception handler, execute at Status.EXL==0 and Status.ERL==0 after the TLB exception handler stores to EPC, Cause, and Status registers.

Pending bus errors can be confirmed by referring to Status.BEM.

2.2 Bus errors are masked when Status.ERL==1 or Status.EXL = 1 (FLX02)

2.2.1 Phenomenon

Even if a bus error occurs during instruction fetch in an exception handler (Status.EXL==1 or Status.ERL==1), the CPU does not accept the exception and executes instruction code with indeterminate values read from the bus.

2.2.2 Corrective measures

This is caused by bus error exceptions being masked by Status.EXL==1 or Status.ERL==1. Do not cause exceptions due to instruction fetch in Status.EXL==1 or Status.ERL==1. Generating exceptions in an exception handler is dangerous. For example:

1) The JR instruction may potentially cause an address error or a bus error. Do not use JR instruction in Status.EXL==1 or Status.ERL==1.

2) A mapped region may potentially cause a TLB exception. Be sure to execute using an unmapped region like that below:
   0x8000_0000 – 0x9FFF_FFFF: kseg0
   0xA000_0000 – 0xBFFF_FFFF: kseg1

2.3 AdEL occurs in index-type ICACHE or BTAC CACHE instructions (FLX03)

2.3.1 Phenomenon

When executing index-type CACHE instructions below in either the User mode or Supervisor mode, operation occasionally becomes undefined and generates AdEL (Address Error exception; load and inst fetch).

There are five index-type ICACHE sub operations as listed below.

00111	CACHE IXIN	I$ index invalidate
00000	CACHE IXLTG	I$ index load tag
00100	CACHE IXSTG	I$ index store tag
00001	CACHE IXLDT	I$ index load data
00101	CACHE IXSDT	I$ index store data

There are four BTAC CACHE sub operations as listed below.

00010	CACHE BXLBT	index load BTAC
00110	CACHE BXSBT	index store BTAC
01100	CACHE BFH	BTAC flush
01010	CACHE BHINBT	hit invalidate BTAC

However, there is no problem when Status.KSU==Kernel. Please note that Status.KSU==Kernel includes the kernel mode at Status.EXL==1 or Status.ERL==1 as well. There is also no problem when Status.CU[0]==0, and Status.KSU==User mode or Supervisor mode.

2.3.2 Corrective measures

In Status.CU[0]==1 and Status.KSU==Supervisor or User, execute under VA[31]==0 when executing either index-type ICACHE or BTAC CACHE instructions. VA here represents base reg + offset.

2.4 kuseg becomes an uncached area when an error exception (Status.ERL = 1) occurs (FLX04)

2.4.1 Phenomenon

There are cases in which kuseg (0x0000_0000 – 0x7FFF_FFFF) becomes uncached in an error exception handler (Status.ERL==1) and data consistency with cached area (kseg, ksseg, kseg0) is lost.

2.4.2 Corrective measures

In an error exception handler (Status.ERL==1), when accessing kuseg (0x0000_0000 – 0x7FFF_FFFF), access it after guarding using SYNC.L as follows:

```
SYNC.L
SW  ku   seg
```

2.5 First two instructions in an exception handler are executed as NOP when a bus error occurs (FLX05)

2.5.1 Phenomenon

There are cases in which the first two instructions in an exception handler are executed as NOP instructions, when certain exception occurs and then a bus error occurs immediately before jumping to the exception handler.

2.5.2 Corrective measures

Place NOP in the first two instruction locations in all exception handlers.

2.6 Unexpected instruction-fetch bus-errors occur when executing a Crashme program (FLX06)

2.6.1 Phenomenon

In Kernerl mode or Supervisor mode, unexpected Instruction-fetch bus errors occur when attempting to execute a program called "Crashme" of Linux, since prohibited instruction-sequences that do not obey the following programming restrictions are executed.

In User mode, such a phenomenon doesn't occur.

2.6.2 Corrective measures

In Kernerl mode or Supervisor mode , obey the following programming restrictions:

1) Any CACHE instruction must not be placed in a branch delay slot.

2) SYNC.P must be located immediately before or immediately after any CACHE instruction.

2. Architecture Overview

This chapter includes an overview of the C790 architecture. It discusses the following items:

- Block diagram and main modules
- Superscalar pipeline operation
- Instruction set
- Registers
- Memory Management
- Cache Memory
- Bus interface
- Floating Point Unit
- Performance Monitors
- Debug Support

2.1 Block Diagram and Functional Block Descriptions

This section presents a block diagram of the main modules of the C790 and summarizes the modules.

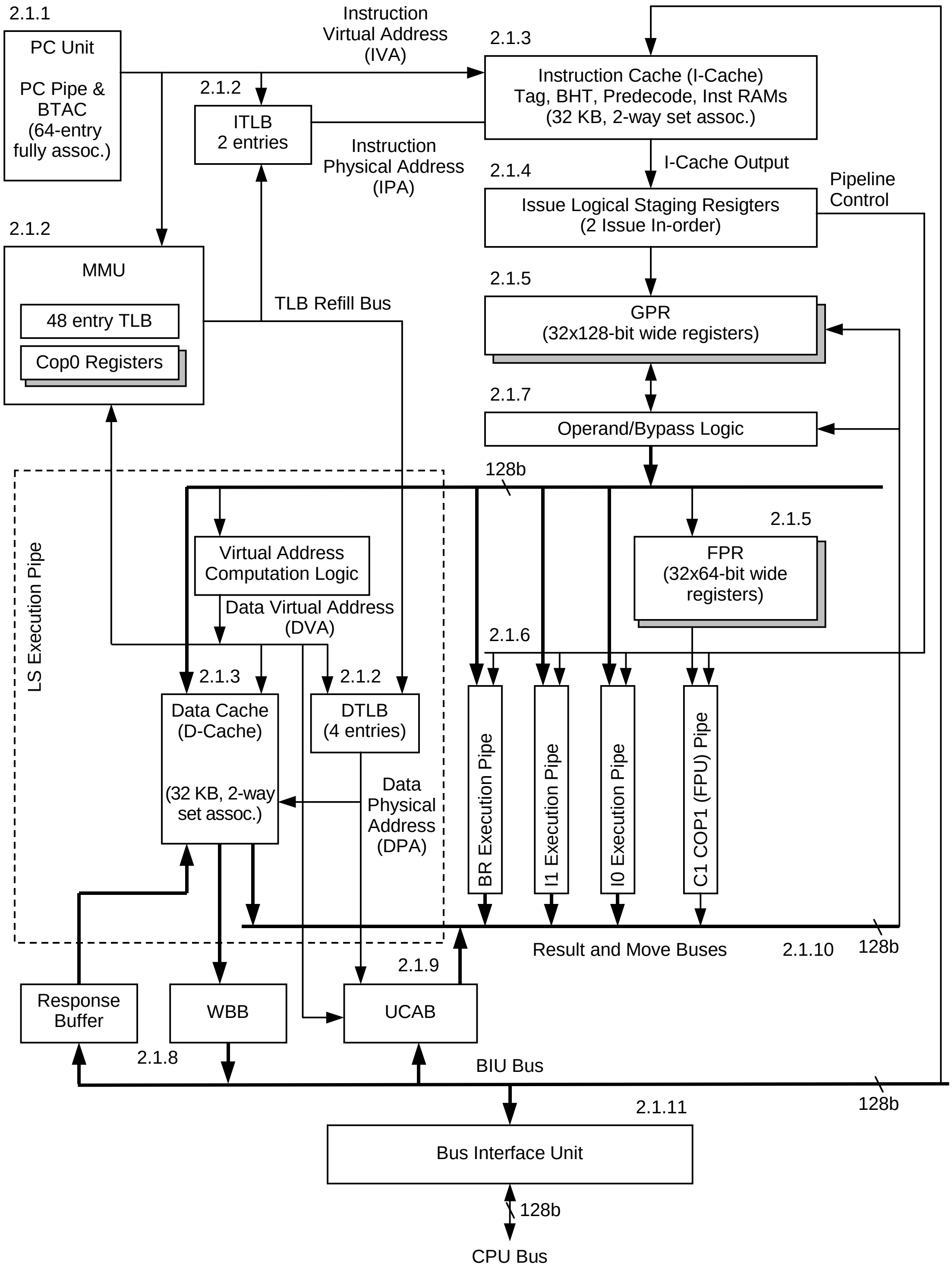


Figure 2-1. C790 Block Diagram

2.1.1 PC Unit

The 32-bit *Program Counter* (*PC*) holds the address of the instruction which is being executed. It also contains a 64-entry **Branch Target Address Cache** (BTAC) which stores branch target addresses used during branch prediction.

2.1.2 MMU

The Memory Management Unit supports the address translation functions of the CPU. It supplies the DTLB (Data Translation Lookaside Buffer) and ITLB (Instruction Translation Lookaside Buffer) with data via the TLB Refill Bus. Usage of these buffers is described in chapter 6.

2.1.3 Caches

Operation of the Instruction Cache and the Data Cache is described in Chapter 7. For each branch instruction, present in the instruction cache, two bits of branch history are stored in the **Branch History Table** (BHT).

2.1.4 Issue Logic and Staging Registers

The issue logic decides how to route instructions to appropriate pipes. It issues up to 2 instructions every cycle. Routing is described and discussed later in section 2.2.

2.1.5 GPR (General Purpose Registers) and FPR (Floating-Point Registers)

The General-Purpose Registers and the Floating-Point Registers are discussed in Section 2.3.

2.1.6 The Five Execution Pipes

2.1.6.1 I0 and I1 Pipes

There are two integer ALU pipelines (I0 and I1), each of which contains a complete 64-bit ALU, Shifter and Multiply-Accumulate unit. The I0 pipeline contains the SA register used for funnel shift operations. **The two 64-bit ALU pipelines can be configured dynamically (on an instruction-by-instruction basis) into a single 128-bit execution pipeline to execute 128-bit Multimedia ALU, Shift and Multiply-Accumulate instructions.** Furthermore, the two ALU pipelines share a single 128-bit multimedia aligner.

2.1.6.2 LS - Load/Store Pipe

The Load/Store (LS) pipe contains logic to support a single 128-bit Load and Store instruction.

2.1.6.3 BR - Branch Pipe

The Branch (BR) pipe contains logic to implement a single Branch instruction including Branch comparators.

2.1.6.4 C1 - COP1/FPU Pipe

The C1 pipe contains logic to support a single/double Floating Point coprocessor unit (COP1).

2.1.7 Operand/Bypass logic

This module takes data from the GPRs and from the Result and Move Buses, and routes the data to the pipelines.

2.1.8 Response Buffer and Writeback Buffer

The Writeback Buffer (WBB) is an 8 entry by 16 byte (one quadword) FIFO queuing up stores prior to accessing the CPU bus. It increases C790 performance by decoupling the processor from the latencies of the CPU bus. It is also used during the gathering operation of uncached accelerated stores; sequential stores less than a quadword in length are gathered in the WBB, thereby reducing bus bandwidth usage.

2.1.9 UCAB

The Uncached Accelerated Buffer (UCAB) is a 1 entry by 8 quadword buffer. It caches 128 sequential bytes of data during an uncached accelerated load miss. Subsequent loads from the uncached accelerated address space get their data from this buffer if the address hits in the UCAB, thereby eliminating bus latencies and providing higher performance.

2.1.10 Result and Move Buses

The Result and Move Buses convey data between execution units, the data cache, and the Operand/Bypass Logic unit.

2.1.11 Bus Interface Unit and BIU Bus

The BIU connects the core to the rest of the system. It interfaces the core's internal bus signals to the CPU Bus.

2.2 Superscalar Pipeline Operation

The C790 has a six-stage superscalar pipeline. It can fetch, decode and execute a maximum of two instructions in parallel each cycle.

This section discusses in more detail the six execution pipelines listed in Section 2.1. It also discusses how instructions are routed among pipes.

2.2.1 Integer Instruction Pipeline Stages

The C790 contains four integer pipelines: the I0 and the I1 pipes, and the Load/Store and Branch pipes. Each pipe consists of the following six stages with each stage having 2 phases:

- I: Instruction Address Select
- Q: Instruction Queue
- R: Register Fetch
- A: Execution
- D: Data Fetch
- W: Write-back

Figure 2-2 shows the six stages of an integer instruction pipeline

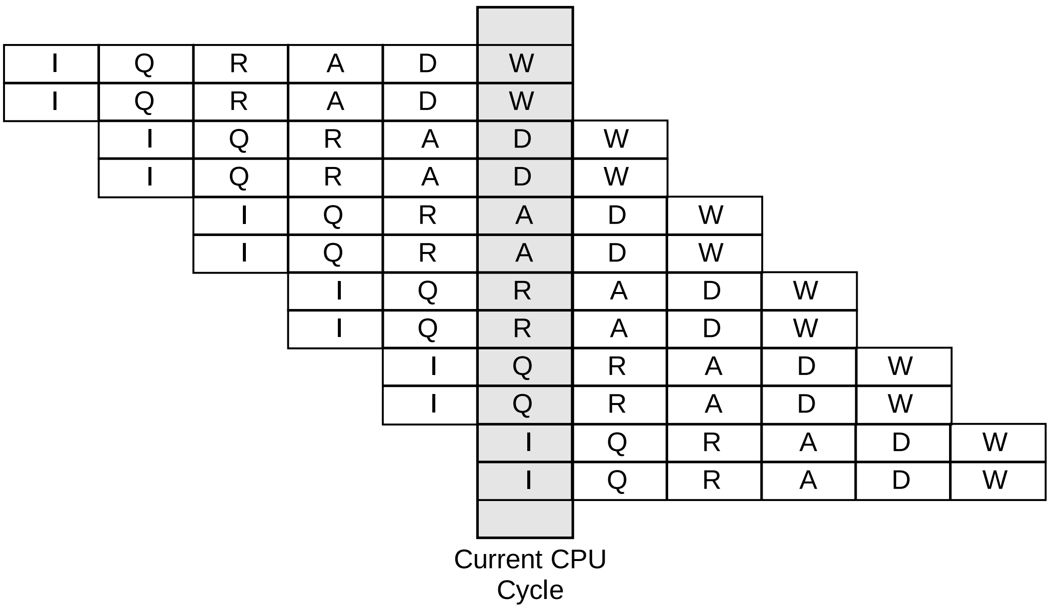


Figure 2-2. C790 Integer Instruction Pipeline

I: Instruction Address Select

During the I stage, the following occurs:

- The sequential address is calculated
- The branch address is calculated
- The instruction address is selected from the following sources
  - Sequential address
  - Actual Branch / Jump address
  - Predicted Branch Target address from the BTAC
  - Exception vector address
  - EPC and Error PC

Q: Instruction Queue

During the Q stage, the following occurs:

- The instruction translation look-aside buffer (ITLB) does the virtual-to-physical address translation
- The instruction cache (data, Tag, steering bits & BHT) fetch begins
- TLB read for instruction fetch starts
- The instruction cache fetch is completed
- TLB read for instruction fetch completes
- The instruction cache Tag hit check is determined and the way selection is done
- The appropriate instructions are selected by the steering bits

R: Register Fetch

During the R stage the following occurs:

- Instructions are bussed to the appropriate execution units
- Register file is read
- Execution unit structural hazards are determined
- Instructions are decoded, data dependencies are determined and the appropriate instructions are issued

A: Execution

During the A stage, the following occurs:

- Results from the D or W stages are bypassed
- The execution units start and complete the integer arithmetic, logical, shift and multimedia instructions
- The iterative steps of the Multiply, Multiply-Accumulate, or Divide instructions are executed
- The virtual address for load and store instructions is calculated
- The branch condition is determined
- The DTLB is read
- The Data Cache and UCAB read starts

D: Data Fetch

During the D stage, the following occurs:

- The TLB read for a data access
- The Data Cache and UCAB read is completed
- The Data Cache Tag checking is completed
- Load or register data is obtained from COP1 (FPU)
- COP0 registers are read
- Data alignment and way selection is done for the data from the Data Cache
- Data sign extension is done
- Complete updating BHT bits and the BTAC
- All the exceptions are detected

W: Write Back

During the W stage, the following occurs:

- For store operations data is written to the Data Cache
- Data for coprocessor data transfer instructions is transferred to COP1 (FPU)
- For register-to-register and load instructions, the result is written to the register file
- COP0, COP1 (FPU) registers are written for coprocessor data transfer instructions

2.2.2 C1 (COP1/FPU) Instruction Pipeline Stages

The C790's C1 (COP1/FPU) pipeline consists of the following eight stages:

- I: Instruction Address Select
- Q: Instruction Queue
- R: Register Fetch
- T: COP1 Register Fetch
- X: FP Execution 1st Stage
- Y: FP Execution 2nd Stage
- Z: FP Execution 3rd Stage
- S: Register File Write Stage

The eight stages of the pipeline for COP1/FPU are shown in Figure 2-3 with some pipeline stages identified with two letters. COP1 instructions execute simultaneously in the main integer pipeline I0 and the coprocessor 1 pipeline. The first letter identifies the main integer pipeline stage and the second letter identifies the coprocessor pipeline stage.

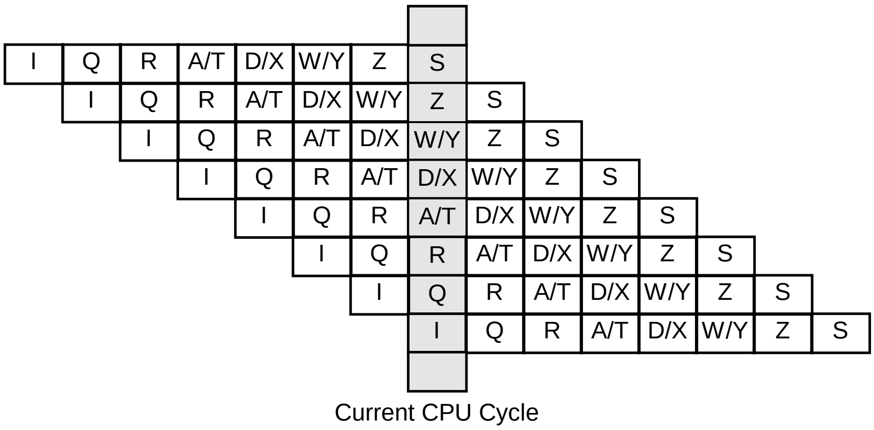


Figure 2-3. FPU Pipeline

The I, Q, and R stages were previously described in Section 2.2.1. The following describes stages specific to the COP1 pipeline:

T: COP1 Register Fetch

During the T stage, the following occurs:

- Register file read for operands
- Bypass muxes from the S Stage/W Stage for S/T overlap.

X: FP Execution 1st Stage

This stage is the first step for floating point operations.

During the X stage, the following occurs:

- Detect Exceptions for input data.
- Detect Exception possibilities for result.
- The Booth function/Wallace multiplication is performed for multiply, the de-nor-malization is performed for add/subtract.

Y: FP Execution 2nd Stage

This stage is the second step for floating point operations. The following occurs:

- Test overflow/underflow on exponent is done
- Normalization for multiplication is done.
- Add/subtract the significand for add/subtract operations.
- Count leading zeros, to determine the shift amount for the normalization

Z: FP Execution 3rd Stage

This stage is the third step for floating point operations. The following occurs:

- Overflow/underflow detection
- Exponent readjustment
- Shift the significand for normalization
- Round the result
- Detect inexact exception

S: Register File Write Stage

During the S stage, the following occurs:

- FPR registers are written.
- FCSR31 is updated.
- Bypass values are passed to the T stage.

2.2.3 Classification and Routing of Instructions According to Execution Pipelines

This section discusses how the five execution pipelines are used in conjunction with instruction routing. Figure 2-4 identifies the specific execution pipelines into which instructions of a particular class are routed, and shows which physical execution units handle instructions from a particular logical pipe. Instruction categories are identified in *italics*, and are shown within the physical pipes where they are executed. ALU instructions can be executed in either integer pipe I0 or I1. COP1 Operate, and COP1 Move instructions execute in two pipes as shown, as does the Wide Operate.

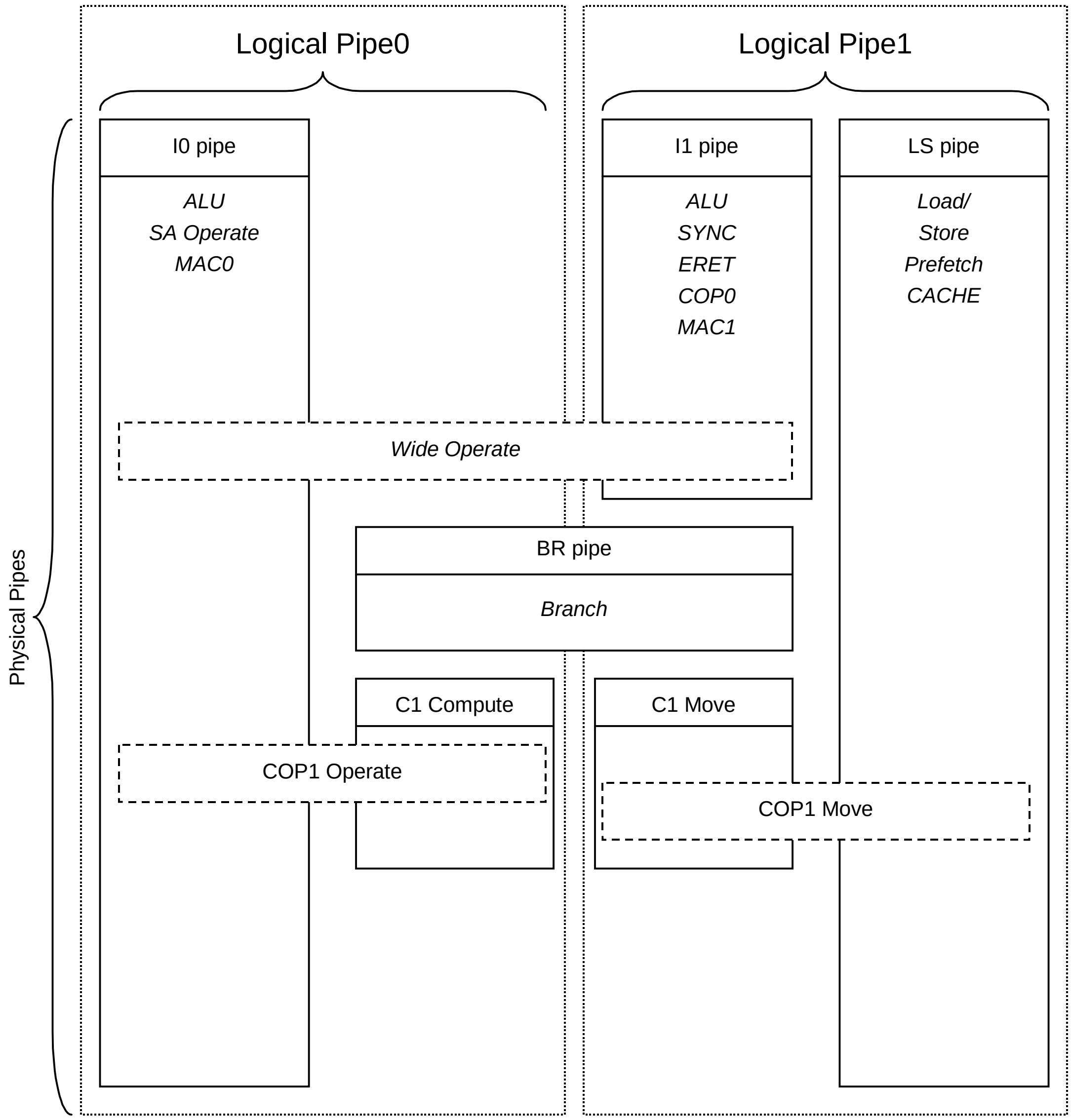


Figure 2-4. Instruction Routing in Logical Pipes and Physical Pipes

Table 2-1 shows the categories of instructions and the execution pipelines that can execute those instructions. The instructions in a single category have the same issuing policy. Instructions which require more than a single execution pipeline are identified in the pipeline column with the (✔&) symbol. For example, COP1 Move requires both the LS and the C1 execution pipelines. On the other hand, the ALU instructions can be executed in either the I0 or the I1 execution pipelines.

Table 2-1. Categories of Instructions and How They Are Routed

Categories	Execution Pipeline					Instructions
	I0	I1	LS	BR	C1	
Load/Store			✔			Load, Store, Wide Load , Wide Store, Prefetch, CACHE
SYNC		✔				Synchronization
ERET		✔				Exception return
SA Operate	✔					Move to/from to SA register
COP0			✔			COP0 Coprocessor move, COP0 Coprocessor operations
COP1 Move[1]			✔&		✔	COP1 Coprocessor move, COP1 Coprocessor Load/Store
COP1 Operate[2]	✔&				✔	COP1 Operate Instructions
ALU[3]	✔	✔				Arithmetic, Shift, Logical, Trap, SYSCALL, BREAK
MAC0	✔					Multiply and Multiply -Accumulate for HI/LO register, MFHI/LO, MTHI/LO
MAC1		✔				Multiply and Multiply-Accumulate for HI1/LO1 register, MFHI1/LO1, MTHI1/LO1
Branch				✔		Branch, Jump, Jump/Link, All Coprocessor Branches
Wide Operate[4]	✔	✔&				Wide ALU, Wide shift, Wide MAC, Funnel shift, Wide HI/LO Moves

[1] COP1 Move instructions execute concurrently in the LS and the C1 pipes.
[2] COP1 Operate instructions execute concurrently in the I0 and the C1 pipes.
[3] ALU instructions can be executed in either the I0 or the I1 pipes.
[4] Wide Operate instructions execute concurrently in the I0 and the I1 pipes.

2.2.4 Instruction Issue Combinations

The C790 always fetches two instructions. A pair of staging registers acts as a 'bellows' between the Q and the R stage. If an instruction can't be issued in a particular cycle, it is saved in the staging registers. In the next cycle the C790 again fetches two instructions and tries to issue two (the one left over in the staging register from the previous cycle and the next sequential one from the pair that is fetched). So the C790 always tries to issue two instructions each cycle whenever it can.

The two instructions that get issued go to the R-stage of the pipeline and get associated with one of two logical pipes: Pipe0 and Pipe1. The instructions are then routed to an appropriate physical pipe for processing.

Instruction categories that can get issued to logical Pipe0 are:

1. ALU
2. Branch
3. Wide Operate
4. SA Operate
5. MAC0
6. COP1 Operate

An alternate way to view this is to recognize that logical Pipe0 is made up of the I0, C1 and BR execution pipelines. When issuing Wide Operate instructions logical Pipe0 also uses the I1 execution pipeline.

Instruction categories that can get issued to logical Pipe1 are:

1. ALU
2. Branch
3. SYNC
4. ERET
5. Load/Store
6. COP1 Move
7. COP0
8. MAC1

An alternate way to view this is to recognize that logical Pipe1 is made up of the I1, LS, C1 and BR execution pipelines.

All instruction categories are statically bound to a single logical pipe, that is, they can only be issued to a particular logical pipe. However the ALU and Branch instruction categories can get issued to either of the two logical pipes. Thus the binding of these two instruction categories to a particular logical pipe is done at instruction issue time.

There are some special cases of instruction sequences that are not allowed in the MIPS ISA. An instruction from the Branch category is not allowed to have another instruction from either the Branch or ERET category in its branch delay slot. So the following pairs of instructions are illegal and effectively never issued together:

1. Branch - Branch
2. Branch - ERET

The following sequences of instructions are also not allowed in the C790. Branch-Likely instructions are a subset of the Branch category (limited to the branch likely instructions).

1. Branch - SYNC.P
2. Branch - SYNC.L
3. Branch - CACHE *1
4. Branch-Likely - MTSA
5. Branch-Likely - MTSAB
6. Branch-Likely - MTSAH
7. Branch-Likely - TLBR *2
8. Branch-Likely - TLBWI *2
9. Branch-Likely - TLBWR *2

*1 CACHE instruction must be guarded by Sync instructions.

Sync.P		Sync.L
CACHE I$	or	CACHE D$
Sync.P		Sync.L

*2 TLBR, TLBWI, TLBWR instructions must be followed by Sync.P

TLBxx
Sync.P

The following table shows the instruction categories which can be issued concurrently to the two logical pipes. All combinations are legal except the ones marked with an “X”. The combinations marked with a “Y” can be issued concurrently, i.e., enter the R stage together but then the younger instruction stalls in the A stage for a single cycle in order to avoid a resource hazard.

Table 2-2. Concurrently Issued Instruction Categories

LOGICAL PIPE1 \ LOGICAL PIPE0	SA Oper.	COP1 Oper.	ALU	MAC0	Branch	Wide Oper.
Load/Store						
ERET					X	
SYNC						
LZC						Y
COP1 Move						
ALU						Y
MAC1						Y
Branch					X	
COP0						

X: illegal combination
Y: Can be issued concurrently but it will stall due to structure hazard.

2.3 Registers

The C790 extends the normal MIPS compatible register set by extending the **general purpose registers** (**GPR**s) from 64-bits to 128-bits, adding an additional pair of HI/LO registers for the I1 pipe and adding the SA register for the funnel shift instruction.

2.3.1 CPU Registers

The C790 has 128-bit wide GPRs. The upper 64 bits of the GPRs are only used by the C790-specific “Quad Load/Store”, and “Multimedia (Parallel)” instructions.

The HI1 and LO1, which are the upper 64 bits of each of the 128-bit HI and LO registers, are also used by new multiply and divide instructions, such as *MULT1*, *MULTU1*, *DIV1*, *DIVU1*, *MADD1*, *MADDU1*, *MFHI1*, *MFLO1*, *MTHI1*, and *MTLO1*, which are non-parallel I1 pipeline-specific instructions.
The SA register contains the shift amount used by the 256 bit funnel shift instruction.

2.3.2 FPU Registers

The floating point unit (COP1) has 64-bit wide floating point registers. It also contains 2 floating point control registers .

2.3.3 COP0 Registers

Table 2-3 identifies the COP0 registers of the C790.

Table 2-3. Coprocessor 0 Registers

Register No.	Register Name	Description	Purpose
0	Index	Programmable register to select TLB entry for reading or writing	MMU
1	Random	Pseudo-random counter for TLB replacement	MMU
2	EntryLo0	Low half of TLB entry for even PFN (Physical page number)	MMU
3	EntryLo1	Low half of TLB entry for odd PFN (Physical page number)	MMU
4	Context	Pointer to kernel virtual PTE table	Exception
5	PageMask	Mask that sets the TLB page size	MMU
6	Wired	Number of wired TLB entries	MMU
7	(Reserved)	Undefined	Undefined
8	BadVAddr	Bad virtual address	Exception
9	Count	Timer compare	Exception
10	EntryHi	High half of TLB entry(Virtual page number and ASID)	MMU
11	Compare	Timer compare	Exception
12	Status	Processor Status Register	Exception
13	Cause	Cause of the last exception taken	Exception
14	EPC	Exception Program Counter	Exception
15	PRId	Processor Revision Identifier	MMU
16	Config	Configuration Register	MMU
17	(Reserved)	Undefined	Undefined
18	(Reserved)	Undefined	Undefined
19	(Reserved)	Undefined	Undefined
20	(Reserved)	Undefined	Undefined
21	(Reserved)	Undefined	Undefined
22	(Reserved)	Undefined	Undefined
23	BadPAddr	Bad Physical Address	Exception
24	Debug	This is used for Debug function	Debug
25	Perf	Performance Counter and Control Register	Exception
26	(Reserved)	Undefined	Undefined
27	(Reserved)	Undefined	Undefined
28	TagLo	Cache Tag register(low bits)	MMU
29	TagHi	Cache Tag register(high bits)	MMU
30	ErrorPC	Error Exception Program Counter	Exception
31	(Reserved)	Undefined	Undefined

2.4 Memory Management

The C790 processor provides a memory management unit (MMU) which uses an on-chip translation look-aside buffer (TLB) to translate virtual addresses into physical addresses.

The C790 supports the MIPS compatible *32-bit* address and *64-bit* data mode. *Only* 32-bit virtual and physical addresses have been implemented. There is no requirement for address sign extension. Address error exception checking will not be done on the “upper” 32-bits (which are ignored). The only condition that will generate the address error exception will be address alignment errors and segment protection errors. In Kernel mode, it is free from address error exception for program counter to wrap-around from *kseg3* to *kuseg*.

Since there is only one addressing mode, all the four MIPS ISAs (I, II, III, IV) and the C790 specific ISA are available without any restrictions in all of the three processor modes (with the appropriate MIPS ISA coprocessor usable restrictions). As such the reserved instruction (RI) exception will occur only when the processor really tries to execute an undefined opcode.

Features

- MIPS III-compatible 32-bit MMU
- Operating Modes: User, Supervisor, and Kernel
- TLB: 48 entries of even/odd page pairs (96 pages)
Fully associative
- Page Size: 4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, 16 MB
- ITLB: 2 entries
- DTLB: 4 entries
- Address Sizes: Virtual Address Size = 32 bit, 2 Gbyte per user Process
Physical Address Size = 32 bit, 4 Gbyte

2.5 Cache Memory

The C790 core contains both an instruction cache and a separate data cache.

Features

The following are the main features of the caches:

- Separate Instruction Cache and Data Cache
- Virtually indexed and physically tagged caches
- Write-back policy for the Data Cache
- Data Cache and Instruction Cache burst read sequential ordering
- Cache Size: Instruction Cache: 32 KB
  Data Cache: 32 KB
- Line Size: 64 Bytes
- Refill size: 64 Bytes
- Associativity: 2-way set-associative
- Write Policy: Write-back and write allocate
- Data order for block reads: Sequential ordering
- Data order for block writes: Sequential ordering
- Instruction cache miss restart: After all data received
- Data cache miss restart: Early restart on first quadword
- Cache parity: No
- Cache Locking: Data Cache Line Lock.
  Controlled by CACHE instruction
- Cache Snooping: No
- Non-blocking load: Yes
- Hit Under Miss: Yes (Multiple hits under one miss are supported)
- Data Cache Prefetch: Yes

2.6 Bus Interface

The C790 CPU core is connected to the rest of the system, and to external devices, through the group of on-chip C790 system bus signals called the CPU Bus.

Features

- Separate data and address buses (Demultiplexed operation)
- 128-bit data bus
- Clocked synchronous operations
- Peak transfer rate of 2.1 GB/sec (@133 MHz bus clock)
- 8/16/32/64/128-bit and burst accesses
- Multimaster capability
- Pipelined operations
- No turn-around or dead cycles between transfers

The CPU Bus does not provide:

- Cache coherency support
- Split transactions

2.7 Floating Point Unit

The floating point unit is IEEE754-1985 compatible as same as FPU in the TX49HF CPU core.

Main Features:

- Tightly coupled to the C790 Integer pipeline.
- Supports both double and single precision format as defined in IEEE-754 specification
- No hardware support for Denormalized number in the IEEE-754 specification. Software (exception handler) supports it.
- The FPU supports five IEEE exceptions and one MIPS defined exception.
- *ADD*, *SUB*, *MUL*, *DIV*, *ABS*, *MOV*, *NEG*, *SQRT*, compare and convert are supported

2.8 Performance Counter

The performance counter provides the means for gathering statistical information about the internal events of the CPU and the pipeline during program execution. The statistics gathered during program execution aid in tuning the performance of hardware and software systems based on the processor.

The performance counter consists of one control register and two counters. The control register controls the functions of the performance counter while the counters count the number of events specified by the control register.

Features:

- Two performance counter registers
- Over twenty different events within the processor can be counted
- Counting can be selectively enabled in User, Supervisor, Kernel, and Exception modes

2.9 Debug and Tracing Functions

The C790 supports real-time PC tracing. Pipeline status, target addresses of indirect jumps, and exception vectors are made available on special signals. The executed instruction sequence can be restored from signals and the source program.

Features:

- One Instruction Address Breakpoint register
- One Instruction Address Breakpoint Mask register
- One Data Address Breakpoint register
- One Data Address Breakpoint Mask register
- One Data Value Breakpoint register
- One Data Value Breakpoint Mask register
- Each breakpoint individually enabled
- Breakpoint function can be selectively enabled in User, Supervisor, Kernel, and Exception modes
- External Trigger signal can be generated when breakpoint occurs
- 11 signals used to provide real-time PC tracing function

3. Instruction Set Overview and Summary

This chapter provides an overview of the C790 instruction set. Refer to Appendices A - D for detailed descriptions of individual instructions.

3.1 Introduction

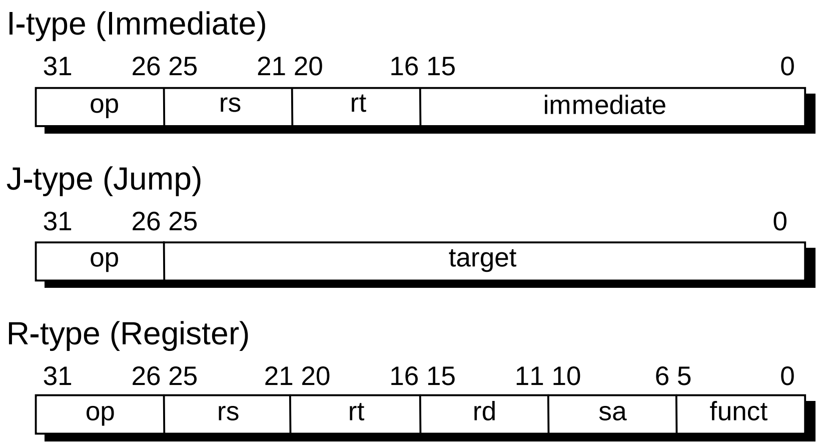
The C790 supports all MIPS III instructions with the exception of 64-bit multiply, 64-bit divide, Load Linked and Store Conditional instructions. It also supports a limited number of MIPS IV instructions and additional C790-specific instructions, such as Multiply/Add instructions and multimedia instructions.

The instruction set can be divided into the following groups:

- Load and Store
- Computational
- Jump and Branch
- Miscellaneous
- System Control Coprocessor (COP0)
- Coprocessor 1 (COP1)
- C790-specific

3.2 CPU Instruction Set Formats

There are three instruction formats: ***immediate*** (I-type), ***jump*** (J-type), and ***register*** (R-type), as shown in Figure 3-1. The use of a small number of instruction formats simplifies instruction decoding (thus producing higher frequency operations) and allows the compiler to synthesize more complicated (and less frequently used) operations and address modes from these three formats as needed.



op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target (source/destination) register or branch condition
immediate	16-bit immediate value, branch displacement or address displacement
target	26-bit jump target address
rd	5-bit destination register specifier
sa	5-bit shift amount
funct	6-bit function field

Figure 3-1. CPU Instruction Formats

3.3 Instruction Set Summary

The C790 supports MIPS III instructions[1] as well as a limited number of MIPS IV instructions. A large number of C790-specific instructions, such as multiply/add instructions and multimedia instructions have also been implemented.

3.3.1 Load/Store Instructions

The instructions in this group transfer data of different sizes: bytes, halfwords, words, doublewords and quadwords. Signed and unsigned integers of different sizes are supported by loads that either sign-extended or zero-extended the data loaded into the register.

Load and store instructions are immediate (I-type) instructions that move data between memory and the general registers. The only addressing mode that load and store instructions directly support is base register plus 16-bit signed immediate offset.

3.3.1.1 Normal Loads and Stores

The C790 does not support Load Linked and Store Conditional instructions, LL, LLD, SC and SCD. For details of these instructions refer to Appendix A.

Table 3-1. Load / Store Instructions

Mnemonic	Description	Defined in
LB	Load Byte	MIPS I
LBU	Load Byte Unsigned	MIPS I
LD	Load Doubleword	MIPS III
LDL	Load Doubleword Left	MIPS III
LDR	Load Doubleword Right	MIPS III
LH	Load Halfword	MIPS I
LHU	Load Halfword Unsigned	MIPS I
LW	Load Word	MIPS I
LWL	Load Word Left	MIPS I
LWR	Load Word Right	MIPS I
LWU	Load Word Unsigned	MIPS III
SB	Store Byte	MIPS I
SD	Store Doubleword	MIPS III
SDL	Store Doubleword Left	MIPS III
SDR	Store Doubleword Right	MIPS III
SH	Store Halfword	MIPS I
SW	Store Word	MIPS I
SWL	Store Word Left	MIPS I
SWR	Store Word Right	MIPS I

[1] Note: The C790 does not support the following MIPS III instructions:
64-bit multiply and divide instructions (DMULT, DMULTU, DDIV, DDIVU)
Semaphore instructions (LL, LLD, SC, SCD)

3.3.1.2 Multimedia Loads and Stores

The C790 implements 128-bit (quadword) load and store instructions for multimedia purpose. For details of these instructions refer to Appendix B.

Table 3-2. Multimedia Load / Store Instructions

Mnemonic	Description	Defined in
LQ	Load Quadword	C790
SQ	Store Quadword	C790

3.3.1.3 Coprocessor Loads and Stores

These loads and stores are coprocessor instructions. A particular coprocessor is enabled if corresponding CU bit is set in CP0 Status register. Otherwise executing one of these instructions generates a Coprocessor Unusable exception. For details of these instructions refer to Appendices C and D.

Table 3-3. Coprocessor Load / Store Instructions

Mnemonic	Description	Defined in
LDC1	Load Doubleword to Floating Point	MIPS II
LWC1	Load Word to Floating Point	MIPS I
SDC1	Store Doubleword from Floating Point	MIPS II
SWC1	Store Word from Floating Point	MIPS I

3.3.1.4 Data Formats and Addressing

The C790 processor uses five data formats:

- 128-bit quadword
- 64-bit doubleword
- 32-bit word
- 16-bit halfword
- 8-bit byte

Byte ordering within each of the larger data formats — halfword, word, doubleword — can be configured in either big-endian or little-endian order. Endianness refers to the location of byte 0 within the multi-byte data structure. Figure 3-2 and Figure 3-3 show the ordering of bytes within words and the ordering of words within multiple-word structures for the big-endian and little-endian conventions.

When the C790 processor is configured as a big-endian system, byte 0 is the most-significant (leftmost) byte, thereby providing compatibility with MC 68000® and IBM 370® conventions. Figure 3-2 shows this configuration.

Higher Address ⇧ Lower Address

Bit #

Word Address	31 24	23 16	15 8	7 0
12	12	13	14	15
8	8	9	10	11
4	4	5	6	7
0	0	1	2	3

Figure 3-2. Big-Endian Byte Ordering

When configured as a little-endian system, byte 0 is always the least-significant (rightmost) byte, which is compatible with iAPX® x86 and DEC VAX® conventions.

Higher Address ⇧ Lower Address

Bit #

Word Address	31 24	23 16	15 8	7 0
12	15	14	13	12
8	11	10	9	8
4	7	6	5	4
0	3	2	1	0

Figure 3-3. Little-Endian Byte Ordering

In this text, bit 0 is always the least-significant (rightmost) bit: thus, bit designations are always little-endian (although no instructions explicitly designate bit positions within words).

Figure 3-4 and Figure 3-5 show little-endian and big-endian byte ordering in doublewords.

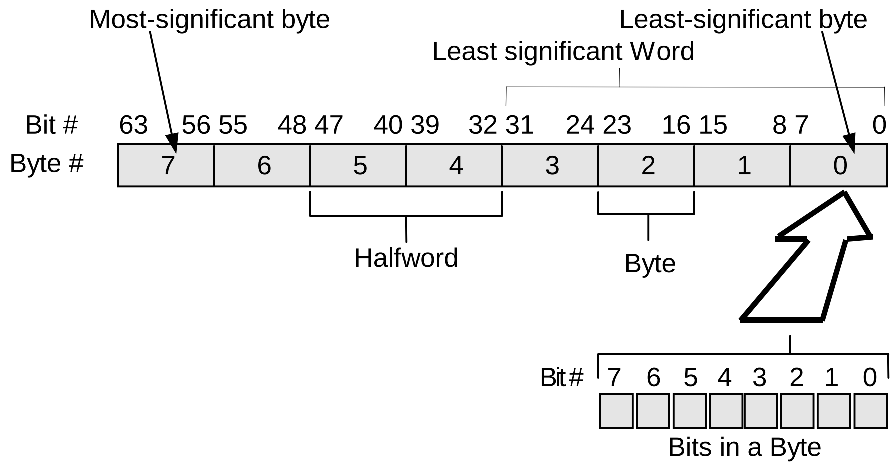


Figure 3-4. Little-Endian Data in a Doubleword

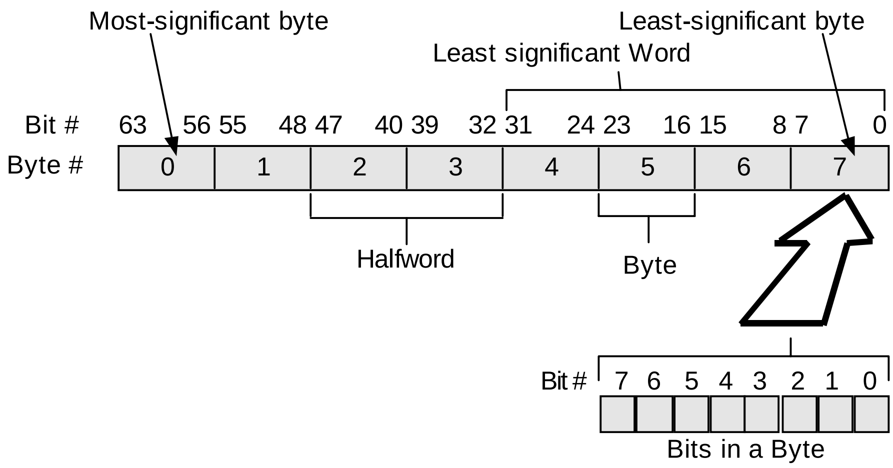


Figure 3-5. Big-Endian Data in a Doubleword

The CPU uses byte addressing for halfword, word, doubleword, and **quadword** accesses with the following alignment constraints:

- Halfword accesses must be aligned on an even byte boundary (0, 2, 4...).
- Word accesses must be aligned on a byte boundary divisible by four (0, 4, 8...).
- Doubleword accesses must be aligned on a byte boundary divisible by eight (0, 8, 16...).
- Quadword accesses must be aligned on a byte boundary divisible by sixteen (0, 16, 32...).

The following special instructions load and store words that are not aligned on 4-byte (word), 8-byte (doubleword), boundaries:

LWL LWR SWL SWR

LDL LDR SDL SDR

These instructions are used in pairs to provide addressing of misaligned words. Addressing misaligned data incurs one additional instruction cycle over that required for addressing aligned data. This extra cycle is because of an extra instruction for the "pair" (e.g.,LWL and LWR form a pair). Also note that the CPU moves the unaligned data at the same rate as a hardware mechanism.

Figure 3-6 and Figure 3-7 shows the access of a misaligned word that has byte address 3.

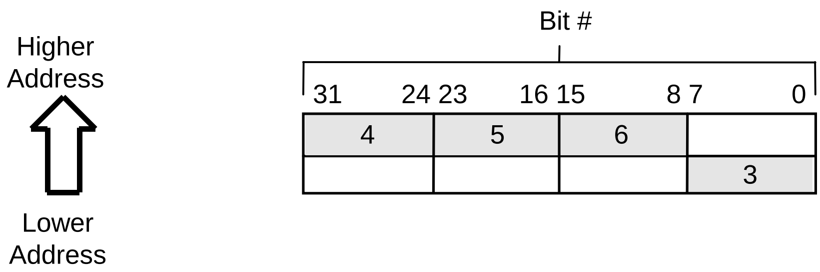


Figure 3-6. Big-Endian Misaligned Word Addressing

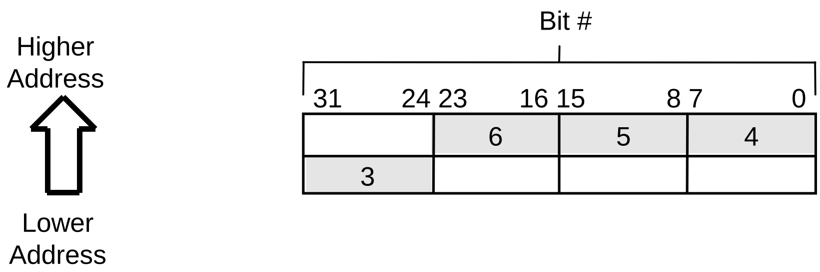


Figure 3-7. Little-Endian Misaligned Word Addressing

3.3.1.5 Defining Access Types

Access type indicates the size of the C790 processor data item to be loaded or stored, set by the load or store instruction opcode.

Regardless of access type or byte ordering (endianess), the address given specifies the low-order byte in the addressed field. For a big-endian configuration, the low-order byte is the most-significant byte; for a little-endian configuration, the low-order byte is the least-significant byte.

The access type, together with the four low-order bits of the address, defines the bytes accessed within the addressed doubleword (shown in Table 3-4 and Table 3-5). Only the combinations shown in Table 3-4 and Table 3-5 are permissible; other combinations cause address error exceptions.

Table 3-4. Defining Access Types (Big-Endian)

Access Type Mnemonic	Low-Order Address Bits				Bytes Accessed Big endian (127---------------95----------------63-----------------31-----------------0) Byte															
	3	2	1	0																
Quadword	0	0	0	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Doubleword	0	0	0	0	0	1	2	3	4	5	6	7								
	1	0	0	0									8	9	10	11	12	13	14	15
Septibyte	0	0	0	0	0	1	2	3	4	5	6									
	0	0	0	1		1	2	3	4	5	6	7								
	1	0	0	0									8	9	10	11	12	13	14	
	1	0	0	1										9	10	11	12	13	14	15
Sextibyte	0	0	0	0	0	1	2	3	4	5										
	0	0	1	0			2	3	4	5	6	7								
	1	0	0	0									8	9	10	11	12	13		
	1	0	1	0											10	11	12	13	14	15
Quintibyte	0	0	0	0	0	1	2	3	4											
	0	0	1	1				3	4	5	6	7								
	1	0	0	0									8	9	10	11	12			
	1	0	1	1												11	12	13	14	15
Word	0	0	0	0	0	1	2	3												
	0	1	0	0					4	5	6	7								
	1	0	0	0									8	9	10	11				
	1	1	0	0													12	13	14	15
Triplebyte	0	0	0	0	0	1	2													
	0	0	0	1		1	2	3												
	0	1	0	0					4	5	6									
	0	1	0	1						5	6	7								
	1	0	0	0									8	9	10					
	1	0	0	1										9	10	11				
	1	1	0	0													12	13	14	
	1	1	0	1														13	14	15
Halfword	0	0	0	0	0	1														
	0	0	1	0			2	3												
	0	1	0	0					4	5										
	0	1	1	0							6	7								
	1	0	0	0									8	9						
	1	0	1	0											10	11				
	1	1	0	0													12	13		
	1	1	1	0															14	15

Access Type Mnemonic	Low-Order Address Bits				Bytes Accessed Big endian (127---------------95----------------63-----------------31-----------------0) Byte															
	3	2	1	0																
Byte	0	0	0	0	0															
	0	0	0	1		1														
	0	0	1	0			2													
	0	0	1	1				3												
	0	1	0	0					4											
	0	1	0	1						5										
	0	1	1	0							6									
	0	1	1	1								7								
	1	0	0	0									8							
	1	0	0	1										9						
	1	0	1	0											10					
	1	0	1	1												11				
	1	1	0	0													12			
	1	1	0	1														13		
	1	1	1	0															14	
	1	1	1	1																15

Table 3-5. Defining Access Types (Little-Endian)

Access Type Mnemonic	Low-Order Address Bits				Bytes Accessed Little endian (127---------------95----------------63-----------------31-----------------0) Byte															
	3	2	1	0																
Quadword	0	0	0	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Doubleword	0	0	0	0									7	6	5	4	3	2	1	0
	1	0	0	0	15	14	13	12	11	10	9	8								
Septibyte	0	0	0	0										6	5	4	3	2	1	0
	0	0	0	1									7	6	5	4	3	2	1	
	1	0	0	0		14	13	12	11	10	9	8								
	1	0	0	1	15	14	13	12	11	10	9									
Sextibyte	0	0	0	0											5	4	3	2	1	0
	0	0	1	0									7	6	5	4	3	2		
	1	0	0	0			13	12	11	10	9	8								
	1	0	1	0	15	14	13	12	11	10										
Quintibyte	0	0	0	0												4	3	2	1	0
	0	0	1	1									7	6	5	4	3			
	1	0	0	0				12	11	10	9	8								
	1	0	1	1	15	14	13	12	11											
Word	0	0	0	0													3	2	1	0
	0	1	0	0									7	6	5	4				
	1	0	0	0					11	10	9	8								
	1	1	0	0	15	14	13	12												
Triplebyte	0	0	0	0														2	1	0
	0	0	0	1													3	2	1	
	0	1	0	0										6	5	4				
	0	1	0	1									7	6	5					
	1	0	0	0						10	9	8								
	1	0	0	1					11	10	9									
	1	1	0	0		14	13	12												
	1	1	0	1	15	14	13													
Halfword	0	0	0	0															1	0
	0	0	1	0													3	2		
	0	1	0	0											5	4				
	0	1	1	0									7	6						
	1	0	0	0							9	8								
	1	0	1	0					11	10										
	1	1	0	0			13	12												
	1	1	1	0	15	14														

Access Type Mnemonic	Low-Order Address Bits 3	2	1	0	Bytes Accessed Little endian (127---------------95----------------63-----------------31-----------------0) Byte															
Byte	0	0	0	0																0
	0	0	0	1															1	
	0	0	1	0														2		
	0	0	1	1													3			
	0	1	0	0												4				
	0	1	0	1											5					
	0	1	1	0										6						
	0	1	1	1									7							
	1	0	0	0								8								
	1	0	0	1							9									
	1	0	1	0						10										
	1	0	1	1					11											
	1	1	0	0				12												
	1	1	0	1			13													
	1	1	1	0		14														
	1	1	1	1	15															

3.3.1.6 Scheduling a Load Delay Slot

A load instruction that does not allow its result to be used by the instruction immediately following is called a *delayed load instruction*. The instruction slot immediately following this delayed load instruction is referred to as the *load delay slot*.

In the C790 processor, the instruction immediately following a load instruction can use the contents of the loaded register. In such cases, however, hardware interlocks insert additional clock cycles. Consequently, scheduling load delay slots can be desirable, both for performance and R-Series processor compatibility. However, the scheduling of load delay slots is not absolutely required.

3.3.2 Computational Instructions

The instructions in this group perform two's complement arithmetic, logical operations, or shifts on integers represented in two's complement notation.

Computational instructions can be either in register (R-type) format, in which both operands are registers, or in immediate (I-type) format, in which one operand is a 16-bit immediate.

Computational instructions perform the following operations on register values:

- Arithmetic
- Logical
- Shift
- Multiply
- Divide

These operations fit in the following four categories of computational instructions:

- ALU immediate instructions
- Three-Operand Register-Type instructions
- Shift instructions
- Multiply and Divide instructions

For detailed information of individual instructions, refer to Appendix A.

*__Note__: The C790 does not support 64-bit Multiply and Divide instructions, DMULT, DMULTU, DDIV, and DDIVU.

3.3.2.1 ALU Immediate Instructions

Table 3-6. ALU Immediate Instructions

Mnemonic	Description	Defined in
ADDI	Add Immediate	MIPS I
ADDIU	Add Immediate Unsigned	MIPS I
SLTI	Set on Less Than Immediate	MIPS I
SLTIU	Set on Less Than Immediate Unsigned	MIPS I
ANDI	AND Immediate	MIPS I
ORI	OR Immediate	MIPS I
XORI	Exclusive OR Immediate	MIPS I
LUI	Load Upper Immediate	MIPS I
DADDI	Doubleword Add Immediate	MIPS III
DADDIU	Doubleword Add Immediate Unsigned	MIPS III

3.3.2.2 Three Operand Register-Type Instructions

Table 3-7. Three Operand Register-Type Instructions

Mnemonic	Description	Defined in
ADD	Add	MIPS I
ADDU	Add Unsigned	MIPS I
SUB	Subtract	MIPS I
SUBU	Subtract Unsigned	MIPS I
DADD	Doubleword Add	MIPS III
DADDU	Doubleword Add Unsigned	MIPS III
DSUB	Doubleword Subtract	MIPS III
DSUBU	Doubleword Subtract Unsigned	MIPS III
SLT	Set Less Than	MIPS I
SLTU	Set Less Than Unsigned	MIPS I
AND	AND	MIPS I
OR	OR	MIPS I
XOR	Exclusive OR	MIPS I
NOR	NOR	MIPS I

3.3.2.3 Shift Instructions

Table 3-8. Shift Instructions

Mnemonic	Description	Defined in
SLL	Shift Left Logical	MIPS I
SRL	Shift Right Logical	MIPS I
SRA	Shift Right Arithmetic	MIPS I
SLLV	Shift Left Logical Variable	MIPS I
SRLV	Shift Right Logical Variable	MIPS I
SRAV	Shift Right Arithmetic Variable	MIPS I
DSLL	Doubleword Shift Left Logical	MIPS III
DSRL	Doubleword Shift Right Logical	MIPS III
DSRA	Doubleword Shift Right Arithmetic	MIPS III
DSLL32	Doubleword Shift Left Logical + 32	MIPS III
DSRL32	Doubleword Shift Right Logical + 32	MIPS III
DSRA32	Doubleword Shift Right Arithmetic + 32	MIPS III
DSLLV	Doubleword Shift Left Logical Variable	MIPS III
DSRLV	Doubleword Shift Right Logical Variable	MIPS III
DSRAV	Doubleword Shift Right Arithmetic Variable	MIPS III

3.3.2.4 Multiply and Divide Instructions

These are the standard MIPS instructions for multiply, divide, and move to / from HI / LO registers executed on the I0 pipeline's MAC unit. See also C790-specific Multiply and Divide instructions discussion.

Table 3-9. Multiply and Divide Instructions

Mnemonic	Description	Defined in
MULT	Multiply	MIPS I
MULTU	Multiply Unsigned	MIPS I
DIV	Divide	MIPS I
DIVU	Divide Unsigned	MIPS I
MFHI	Move From HI	MIPS I
MTHI	Move To HI	MIPS I
MFLO	Move From LO	MIPS I
MTLO	Move To LO	MIPS I

3.3.2.5 64-Bit Operations

The result of operations that use incorrect sign-extended 32-bit values for 64-bit operations is unpredictable.

3.3.3 Jump and Branch Instructions

The architecture defines PC-relative conditional branches, a PC-region unconditional jump, an absolute (register) unconditional jump, and a similar set of procedure calls that record a return link address in a general register. For convenience, these are all referred to here as branches.

All branches have an architectural delay of one instruction. When a branch is taken, the instruction immediately following the branch instruction, in the branch delay slot, is executed before the branch to the target instruction takes place. Conditional branches come in two versions that treat the instruction in the delay slot differently when the branch is not taken and execution falls through. The ‘branch’ instructions execute the instruction in the delay slot, but the ‘branch likely’ instructions do not. (They are said to ‘nullify’ it.)

By convention, if an exception or interrupt prevents the completion of an instruction occupying a branch delay slot, the instruction stream is continued by re-executing the branch instruction. To permit this, branches must be restartable; procedure calls may not use the register in which the return link is stored (usually register 31) to determine the branch target address.

For detailed information of individual instructions, refer to Appendix A. Branch on Coprocessor instructions are covered under coprocessor’s discussions.

3.3.3.1 Jump Instructions

Subroutine calls in high-level languages are usually implemented with Jump or Jump and Link instructions, both of which are J-type instructions. In J-type format, the 26-bit target address shifts 2 bits and combines with the high-order 4-bits of the current program counter to form an absolute address.

Returns, dispatches, and large cross-page jumps are usually implemented with the Jump Register or Jump and Link Register instructions. Both are R-type instructions that take the 32-bit byte address contained in one of the general purpose registers.

Table 3-10. Jump Instructions Jumping Within a 256 MByte Region

Mnemonic	Description	Defined in
J	Jump	MIPS I
JAL	Jump and Link	MIPS I

Table 3-11. Jump Instructions to Absolute Address

Mnemonic	Description	Defined in
JR	Jump Register	MIPS I
JALR	Jump and Link Register	MIPS I

3.3.3.2 Branch Instructions

All branch instruction target addresses are computed by adding the address of the instruction in the branch delay slot to the 16-bit offset (shifts left 2 bits and is sign-extended to 32-bits). All branches occur with a delay of one instruction.

In case of a Branch Likely instruction, if a condition is not taken, the instruction in the delay slot is nullified.

Table 3-12. PC-Relative Conditional Branch Instructions Comparing 2 Registers

Mnemonic	Description	Defined in
BEQ	Branch on Equal	MIPS I
BNE	Branch on Not Equal	MIPS I
BLEZ	Branch on Less Than or Equal to Zero	MIPS I
BGTZ	Branch on Greater Than Zero	MIPS I
BEQL	Branch on Equal Likely	MIPS II
BNEL	Branch on Not Equal Likely	MIPS II
BLEZL	Branch on Less Than or Equal to Zero Likely	MIPS II
BGTZL	Branch on Greater Than Zero Likely	MIPS II

Table 3-13. PC-Relative Conditional Branch Instructions Comparing Against Zero

Mnemonic	Description	Defined in
BLTZ	Branch on Less Than Zero	MIPS I
BGEZ	Branch on Greater Than or Equal to Zero	MIPS I
BLTZAL	Branch on Less Than Zero and Link	MIPS I
BGEZAL	Branch on Greater Than or Equal to Zero and Link	MIPS I
BLTZL	Branch on Less Than Zero Likely	MIPS II
BGEZL	Branch on Greater Than or Equal to Zero Likely	MIPS II
BLTZALL	Branch on Less Than Zero and Link Likely	MIPS II
BGEZALL	Branch on Greater Than or Equal to Zero and Link Likely	MIPS II

3.3.4 Miscellaneous Instructions

3.3.4.1 Exception Instructions

Exception instructions have as their sole purpose causing an exception that will transfer control to a software exception handler in the kernel. System call and breakpoint instructions cause exceptions unconditionally. The trap instructions cause exceptions conditionally based upon the result of a comparison. For detail of these instructions, refer to the individual instruction as described in Appendix A.

Table 3-14. Exception Instructions

Mnemonic	Description	Defined in
BREAK	Breakpoint	MIPS I
SYSCALL	System Call	MIPS I
TGE	Trap if Greater or Equal	MIPS II
TGEU	Trap if Greater or Equal Unsigned	MIPS II
TLT	Trap if Less Than	MIPS II
TLTU	Trap if Less Than Unsigned	MIPS II
TEQ	Trap if Equal	MIPS II
TNE	Trap if Not Equal	MIPS II
TGEI	Trap if Greater or Equal Immediate	MIPS II
TGEIU	Trap if Greater or Equal Immediate Unsigned	MIPS II
TLTI	Trap if Less Than Immediate	MIPS II
TLTIU	Trap if Less Than Immediate Unsigned	MIPS II
TEQI	Trap if Equal Immediate	MIPS II
TNEI	Trap if Not Equal Immediate	MIPS II

3.3.4.2 Serialization Instructions

The order in which memory accesses from load and store instructions appear outside the C790 is not specified by the architecture. The SYNC (or SYNC.L) instruction creates a point in the executing instruction stream at which the relative order of some loads and store is known. Loads and stores executed before the SYNC (or SYNC.L) are retired before loads and stores after the SYNC (or SYNC.L) can start.

In order to guarantee the completion of certain instructions a SYNC.P instruction can be used. Instructions executed before a SYNC.P instruction are completed before instructions after the SYNC.P can start. For detail of this instruction refer to SYNC instruction as described in Appendix A.

Table 3-15. Serialization Instructions

Mnemonic	Description	Defined in
SYNC[2]	Synchronization	MIPS II

[2] This includes the SYNC, SYNC.L and SYNC.P instructions.

3.3.4.3 MIPS IV Instructions

The C790 supports a part of the MIPS IV instructions: Conditional Move instructions and Prefetch instruction.

Conditional move operations allow 'IF' statements to be represented without branches. 'THEN' and 'ELSE' clauses are computed unconditionally and the results are placed in a temporary register. Conditional move operations then transfer the temporary results to their true register.

The Prefetch instruction fetches data expected to be used in the near future and places it in the data cache.

For detail of these instructions, refer to the individual instruction as described in Appendix A.

Table 3-16. MIPS IV Instructions

Mnemonic	Description	Defined in
MOVN	Move Conditional on Not Zero	MIPS IV
MOVZ	Move Conditional on Zero	MIPS IV
PREF	Prefetch	MIPS IV

3.3.5 System Control Coprocessor (COP0) Instructions

COP0 instructions perform operations specifically on the System Control Coprocessor registers to manipulate the memory management, exception handling, performance monitor, and debug facilities of the processor.

COP0 instructions are enabled if the processor is in Kernel mode, or if bit 28 (CU) is set in the Status register. Otherwise executing one of these instructions generates a Coprocessor Unusable Exception.

For details of COP0 instructions refer to Appendix C.

Table 3-17. System Control Coprocessor Instructions

Mnemonic	Description	Defined in
BC0F	Branch on Coprocessor 0 False	MIPS I
BC0T	Branch on Coprocessor 0 True	MIPS I
BC0FL	Branch on Coprocessor 0 False Likely	MIPS II
BC0TL	Branch on Coprocessor 0 True Likely	MIPS II
CACHE	Cache Operation	R4000
DI	Disable Interrupt	C790
EI	Enable Interrupt	C790
ERET	Exception Return	R4000
TLBR	Read Indexed TLB Entry	R4000
TLBWI	Write Index TLB Entry	R4000
TLBWR	Write Random TLB Entry	R4000
TLBP	Probe TLB for Matching Entry	R4000
MTC0	Move To System Control Coprocessor	R4000
MFC0	Move From System Control Coprocessor	R4000
MTPC	Move To Performance Counter	C790
MFPC	Move From Performance Counter	C790
MTPS	Move To Performance Event Specifier	C790
MFPS	Move From Performance Event Specifier	C790
MTBPC	Move To Breakpoint Control Register	C790
MFBPC	Move From Breakpoint Control Register	C790
MTDAB	Move To Data Address Breakpoint Register	C790
MFDAB	Move From Data Address Breakpoint Register	C790
MTDABM	Move To Data Address Breakpoint Mask Register	C790
MFDABM	Move From Data Address Breakpoint Mask Register	C790
MTIAB	Move To Instruction Address Breakpoint Register	C790
MFIAB	Move From Instruction Address Breakpoint Register	C790
MTIABM	Move To Instruction Address Breakpoint Mask Register	C790
MFIABM	Move From Instruction Address Breakpoint Mask Register	C790
MTDVB	Move To Data Value Breakpoint Register	C790
MFDVB	Move From Data Value Breakpoint Register	C790
MTDVBM	Move To Data Value Breakpoint Mask Register	C790
MFDVBM	Move From Data Value Breakpoint Mask Register	C790

3.3.6 Coprocessor 1 (COP1)

Coprocessor instructions perform operations in their respective coprocessors. Coprocessor loads and stores are I-type, and coprocessor computational instructions have coprocessor-dependent formats. Coprocessor load and store instructions are summarized in 3.3.1.3.

3.3.6.1 Coprocessor 1 (COP1) Instructions

COP1 instructions are enabled if bit 29 (CU) is set in the Status register. Otherwise executing one of these instructions generates a Coprocessor Unusable Exception. For details of COP1 instructions refer to Appendix D.

Table 3-18. Coprocessor 1 Instructions

Mnemonic	Description	Defined in
BC1F	Branch on Floating Point False	MIPS I
BC1T	Branch on Floating Point True	MIPS I
LWC1	Load Word to Floating Point	MIPS I
LDC1	Load Doubleword to Floating Point	MIPS II
SWC1	Store Word from Floating Point	MIPS I
SDC1	Store Doubleword from Floating Point	MIPS II
MFC1	Move Word from Floating Point	MIPS I
MTC1	Move Word to Floating Point	MIPS I
DMFC1	Move Doubleword from Floating Point	MIPS III
DMTC1	Move Doubleword to Floating Point	MIPS III
CFC1	Move Control Word from Floating Point	MIPS I
CTC1	Move Control Word to Floating Point	MIPS I
CVT.D.fmt	Floating Point Convert to Double Floating Point	MIPS I, III
CVT.L.fmt	Floating Point Convert to Long Fixed Point	MIPS III
CVT.S.fmt	Floating Point Convert to Single Floating Point	MIPS I, III
CVT.W.fmt	Floating Point Convert to Word Fixed Point	MIPS I
ADD.fmt	Floating Point Add	MIPS I
SUB.fmt	Floating Point Subtract	MIPS I
MUL.fmt	Floating Point Multiply	MIPS I
DIV.fmt	Floating Point Divide	MIPS I
ABS.fmt	Floating Point Absolute	MIPS I
MOV.fmt	Floating Point Move	MIPS I
NEG.fmt	Floating Point Negate	MIPS I
SQRT.fmt	Floating Point Square Root	MIPS II
C.cond.fmt	Floating Point Compare	MIPS I
CEIL.L.fmt	Floating Point Ceiling Convert to Long Fixed Point	MIPS III
CEIL.W.fmt	Floating Point Ceiling Convert to Word Fixed Point	MIPS II
FLOOR.L.fmt	Floating Point Floor Convert to Long Fixed Point	MIPS III
FLOOR.W.fmt	Floating Point Floor Convert to Word Fixed Point	MIPS II
ROUND.L.fmt	Floating Point Round to Long Fixed Point	MIPS III
ROUND.W.fmt	Floating Point Round to Word Fixed Point	MIPS II
TRUNC.L.fmt	Floating Point Truncate to Long Fixed Point	MIPS III
TRUNC.W.fmt	Floating Point Truncate to Word Fixed Point	MIPS II

3.3.7 C790-Specific Instructions

The C790 extends its instruction set from the original MIPS architecture. The following instructions are supported:

- Three-operand Multiply and Multiply/Add instructions
- Multiply instructions for Pipeline 1
- Multimedia instructions
- Enable interrupt and Disable interrupt instructions

For more information, refer to Appendices B and C.

3.3.7.1 Integer Multiply / Divide Instructions

The standard MIPS instructions for multiply, divide and move to / from HI / LO registers execute on the I0 pipeline's MAC unit. A complete set of new instructions has also been defined to execute on the I1 pipeline's MAC unit. All of these instructions are shown in the following table.

Table 3-19. C790-Specific Multiply and Divide Instructions

OpCode	Description	OpCode	Description
(Three Operand Multiply and Multiply-add)		DIV1	Divide 1
MADD	Multiply/Add	DIVU1	Divide Unsigned 1
MADDU	Multiply/Add Unsigned	MADD1	Multiply/Add 1
MULT	Multiply(3-operand)	MADDU1	Multiply/Add Unsigned 1
MULTU	Multiply Unsigned(3-operand)	MFHI1	Move From HI 1
(Multiply Instructions for Pipeline 1)		MFLO1	Move From LO 1
MULT1	Multiply 1	MTHI1	Move To HI 1
MULTU1	Multiply Unsigned 1	MTLO1	Move To LO 1

The C790 supports three-operand multiply instructions that store the multiply result to a general purpose register in addition to the LO register. These instructions, as such, don't have to use the MFLO instruction to move data from the LO register to a general purpose register.

- **MULT rd, rs, rt** HI | | LO = rs * rt (signed)
  rd = new LO contents
- **MULTU rd, rs, rt** HI | | LO = rs * rt (unsigned)
  rd = new LO contents

The C790 also supports new multiply-add instructions, MADD and MADDU. These instructions execute multiply-accumulate operations using the HI and LO registers as accumulators.

- **MADD rd, rs, rt** HI | | LO += rs * rt (signed)
  rd = new LO contents
- **MADDU rd, rs, rt** HI | | LO += rs * rt (unsigned)
  rd = new LO contents

3.3.7.2 Multimedia Instructions

The C790 defines a new set of instructions to support multimedia applications. These instructions are shown in Table 3-20. Most of these instructions do parallel operations on data by combining the execution units of the two pipelines (I0 and I1). They form a 128-bit path and then do parallel operations on either two 64-bit data items, four 32-bit data items, eight 16-bit data items, or sixteen 8-bit data items.

In order to support the 128-bit datapath, 128-bit load/store operations are also implemented.

Table 3-20. Multimedia Instructions

OpCode	Description
(Arithmetic)	
PADDB	Parallel Add Byte
PSUBB	Parallel Subtract Byte
PADDH	Parallel Add Halfword
PSUBH	Parallel Subtract Halfword
PADDW	Parallel Add Word
PSUBW	Parallel Subtract Word
PADSBH	Parallel Add/Subtract Halfword
PADDSB	Parallel Add with Signed Saturation Byte
PSUBSB	Parallel Subtract with Signed Saturation Byte
PADDSH	Parallel Add with Signed Saturation Halfword
PSUBSH	Parallel Subtract with Signed Saturation Halfword
PADDSW	Parallel Add with Signed Saturation Word
PSUBSW	Parallel Subtract with Signed Saturation Word
PADDUB	Parallel Add with Unsigned Saturation Byte
PSUBUB	Parallel Subtract with Unsigned Saturation Byte
PADDUH	Parallel Add with Unsigned Saturation Halfword
PSUBUH	Parallel Subtract with Unsigned Saturation Halfword
PADDUW	Parallel Add with Unsigned Saturation Word
PSUBUW	Parallel Subtract with Unsigned Saturation Word
(Min/Max)	
PMAXH	Parallel Maximum Halfword
PMINH	Parallel Minimum Halfword
PMAXW	Parallel Maximum Word
PMINW	Parallel Minimum Word

OpCode	Description
(Absolute)	
PABSH	Parallel Absolute Halfword
PABSW	Parallel Absolute Word
(Multiply and Divide)	
PMULTW	Parallel Multiply Word
PMULTUW	Parallel Multiply Unsigned Word
PDIVW	Parallel Divide Word
PDIVUW	Parallel Divide Unsigned Word
PMADDW	Parallel Multiply/Add Word
PMADDUW	Parallel Multiply/Add Unsigned Word
PMSUBW	Parallel Multiply/Subtract Word
PMFHI	Parallel Move From HI
PMFLO	Parallel Move From LO
PMTHI	Parallel Move To HI
PMTLO	Parallel Move To LO
PMULTH	Parallel Multiply Halfword
PMADDH	Parallel Multiply/Add Halfword
PMSUBH	Parallel Multiply/Subtract Halfword
PMFHL	Parallel Move From HI/LO
PMTHL	Parallel Move To HI/LO
PHMADH	Parallel Horizontal Multiply/Add Halfword
PHMSBH	Parallel Horizontal Multiply/Subtract Halfword
PDIVBW	Parallel Divide Broadcast Word

OpCode	Description
(SA Operation)	
MFSA	Move from SA Register
MTSA	Move to SA Register
MTSAB	Move Byte Count to SA Register
MTSAH	Move Halfword Count to SA Register
(Shift)	
PSLLH	Parallel Shift Left Logical Halfword
PSRLH	Parallel Shift Right Logical Halfword
PSRAH	Parallel Shift Right Arithmetic Halfword
PSLLW	Parallel Shift Left Logical Word
PSRLW	Parallel Shift Right Logical Word
PSRAW	Parallel Shift Right Arithmetic Word
PSLLVW	Parallel Shift Left Logical Variable Word
PSRLVW	Parallel Shift Right Logical Variable Word
PSRAVW	Parallel Shift Right Arithmetic Variable Word
(Logical)	
PAND	Parallel AND
POR	Parallel OR
PXOR	Parallel XOR
PNOR	Parallel NOR
(Compare)	
PCGTB	Parallel Compare for Greater Than Byte
PCEQB	Parallel Compare for Equal Byte
PCGTH	Parallel Compare for Greater Than Halfword
PCEQH	Parallel Compare for Equal Halfword
PCGTW	Parallel Compare for Greater Than Word
PCEQW	Parallel Compare for Equal Word

OpCode	Description
(Quadword Load Store)	
LQ	Load Quadword
SQ	Store Quadword
(Pack/Extend)	
PPACB	Parallel Pack To Byte
PPACH	Parallel Pack To Halfword
PINTEH	Parallel Interleave Even Halfword
PPACW	Parallel Pack To Word
PEXTUB	Parallel Extend Upper From Byte
PEXTLB	Parallel Extend Lower From Byte
PEXTUH	Parallel Extend Upper From Halfword
PEXTLH	Parallel Extend Lower From Halfword
PEXTUW	Parallel Extend Upper From Word
PEXTLW	Parallel Extend Lower From Word
PEXT5	Parallel Extend from 5 bits
PPAC5	Parallel Pack to 5 bits
(Others)	
PCPYH	Parallel Copy Halfword
PCPYLD	Parallel Copy Lower Doubleword
PCPYUD	Parallel Copy Upper Doubleword
PREVH	Parallel Reverse Halfword
PINTH	Parallel Interleave Halfword
PEXEH	Parallel Exchange Even Halfword
PEXCH	Parallel Exchange Center Halfword
PEXEW	Parallel Exchange Even Word
PEXCW	Parallel Exchange Center Word
PROT3W	Parallel Rotate 3 word
QFSRV	Quadword Funnel Shift Right Variable
PLZCW	Parallel Leading Zero Count Word

3.4 User Instruction Latency and Repeat Rate

Table 3-21 shows the latencies and repeat rates for all user instructions executed in I0, I1, BR, LS and C1 execution pipelines. Kernel instructions are not included, nor are instructions not issued to these execution pipelines. See Figure 2-1 and Figure 2-4 for execution pipeline name.

Table 3-21. Latencies and Repeat Rates for User Instruction

Instruction Type	Execution	Latency	Repeat Rate	Comment
Integer Instructions				
Add/Sub/Logical/Set	I0/I1	1	1	
MF/MT/HI/LO	I0/I1	1	1	
Shift/LUI	I0/I1	1	1	
Branch/Jump	BR	1	1	
Conditional Move	I0/I1	1	1	
MULT/MULTU	I0	4	2	Latency relative to Lo/Hi/GPR
MULT1/MULTU1	I1	4	2	Latency relative to Lo1/Hi1/GPR
DIV/DIVU	I0	37	37	Latency relative to Lo/Hi
DIV1/DIVU1	I1	37	37	Latency relative to Lo1/Hi1
MADD/MADDU	I0	4	2	Latency relative to Lo/Hi/GPR
MADD1/MADDU1	I1	4	2	Latency relative to Lo1/Hi1/GPR
Load	LS	1	1	Assuming cache hit
Store	LS	-	1	Assuming cache hit
Multimedia Multiply	I0+I1	4	2	
Multimedia Multiply/Add	I0+I1	4	2	
Multimedia Divide	I0+I1	37	37	
Floating-Point Instructions				
ADD.S/SUB.S/C.cond.S	C1	6	2	
ADD.D/SUB.D/C.cond.D	C1	8	2	
ABS/NEG/MOV	C1	6	2	
CVT	C1	8	2	
MUL.S	C1	6	2	
MUL.D	C1	8	2	
DIV.S	C1	21	15	
DIV.D	C1	35	29	
SQRT.S	C1	21	15	
SQRT.D	C1	35	29	
MFC1/MTC1	C1+LS	2	1	
DMFC1/DMTC1	C1+LS	2	1	
CFC1/CTC1	C1+LS	2	1	
LWC1/LDC1	C1+LS	2	1	Assuming cache hit
SWC1/SDC1	C1+LS	–	1	

4. CPU and COP0 Registers

This chapter describes the CPU registers and the System Control Coprocessor (COP0) registers.

The CPU registers group consists of:

- *General Purpose Registers (GPRs),*
- *Multiply and Divide registers (**HI** and **LO** registers) that hold the results of integer multiply and divide,*
- *The **SA** register which is used by the funnel shift instructions,*
- *The* **Program Counter** *(PC) register.*

The *COP0* registers control the processor state and report its status. These registers can be read using the *MFC0* instruction and written using the *MTC0* instruction.

4.1 CPU Registers

The central processing unit (CPU) provides the following registers:

- *32 128-bit* General Purpose Registers *(*GPR*)*
- *Four registers that hold the results of integer multiply and divide operations (*HI0*,* LO0*,* HI1*, and* LO1*)*
- Shift Amount (SA) *register*
- Program Counter

The C790 has 128-bit-wide General Purpose Registers (GPRs). The upper 64 bits of the GPRs are only used by the C790-specific “Quad Load/Store”, and “Multimedia (Parallel)” instructions.

HI0 and *LO0* are the standard 64-bit *HI* and *LO* registers. *HI1* and *LO1*, which are the upper 64 bits of the 128-bit *HI* and *LO* registers, are only used by the new multiply and divide instructions, such as *MULT1*, *MULTU1*, *DIV1*, *DIVU1*, *MADD1*, *MADDU1*, *MFHI1*, *MFLO1*, *MTHI1*, and *MTLO1*. All these instructions are equivalent to existing instructions which operate on *HI0* and *LO0* registers.

The *Shift Amount* (SA) register specifies the shift amount used by the funnel shift instruction. The shaded registers in Figure 4-1 are new architecturally-visible registers that are specific to the C790.

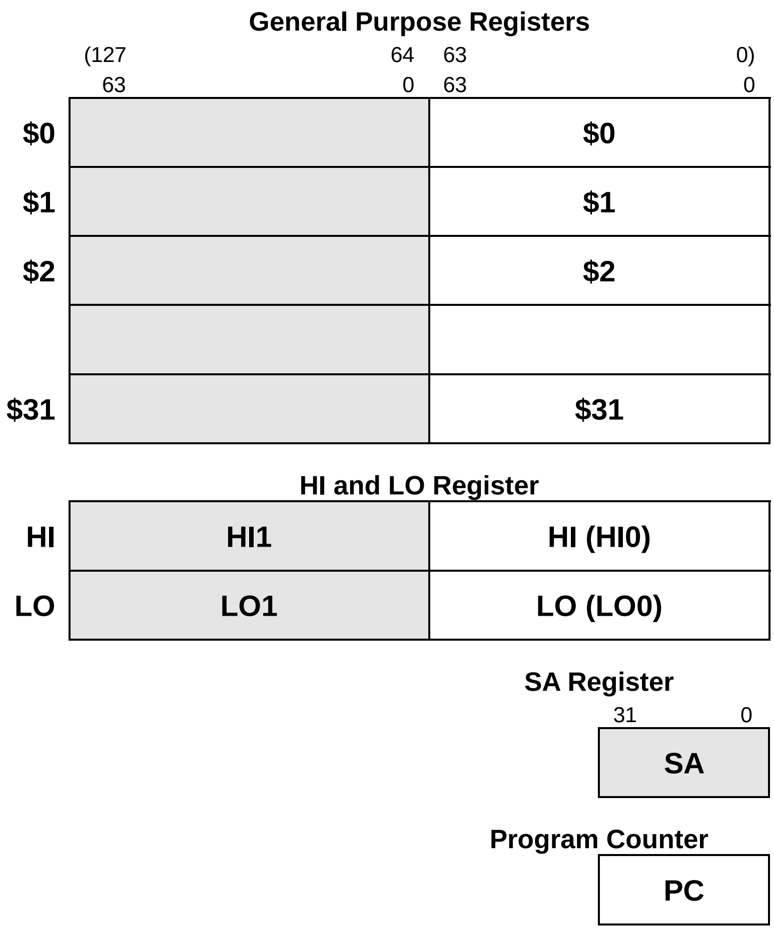


Figure 4-1. CPU Registers

4.1.1 General Purpose Registers

The standard 64-bit CPU general purpose registers have been extended to 128-bit registers. New instructions have been defined to use the upper 64-bits of these registers.

Two of the CPU general purpose registers have special assigned functions:

- *r0 is hardwired to a value of zero, and can be used as the target register for any instruction whose result is to be discarded. r0 can also be used as a source when a zero value is needed.*
- *r31 is the link register used by the Jump and Link instructions. In general, it should not be used by other instructions.*

4.1.2 HI and LO Registers

The standard 64-bit *HI* and *LO* registers have been extended to 128-bit registers. New instructions have been defined to use the upper 64-bits of these registers. *HI0* and *LO0* are the standard 64-bit *HI* and *LO* registers. HI1 and LO1 are the upper 64 bits of the 128-bit *HI* and *LO* registers

These four registers (*HI0*, *LO0*, *HI1*, *LO1*) store:

- *the product of integer multiply operations, or*
- *the accumulation of integer multiply-accumulate operations, or*
- *the quotient (in* LO0 *or* LO1*) and remainder (in* HI0 *or* HI1*) of integer divide operations.*

4.1.3 Shift Amount (SA) Register

The *SA* register specifies the shift amount used by the funnel shift instruction. This is a new architecturally-visible register and it needs to be saved and restored as part of the processor state. New instructions have been defined to move values between this register and the general purpose registers.

4.1.4 Program Counter (PC)

The *Program Counter* (*PC*) holds the address of the instruction which is being executed. The *PC* is incremented automatically by 4 when a non-control-transfer instruction (that is: *branch, jump, ERET, SYSCALL,* or *TRAP*) is executed. Control-transfer instructions change the value of the *PC* to the target address specified by them. An exception also changes the contents of the *PC* to the specified exception vector address.

4.2 System Control Coprocessor (COP0) Registers

COP0 registers are listed in Table 4-1.

Table 4-1. Coprocessor 0 Registers

Register No.	Register Name	Description	Purpose
0	Index	Programmable register to select TLB entry for reading or writing	MMU
1	Random	Pseudo-random counter for TLB replacement	MMU
2	EntryLo0	Low half of TLB entry for even PFN (Physical page number)	MMU
3	EntryLo1	Low half of TLB entry for odd PFN (Physical page number)	MMU
4	Context	Pointer to kernel virtual PTE table in 32-bit addressing mode	Exception
5	PageMask	Mask that sets the TLB page size	MMU
6	Wired	Number of wired TLB entries	MMU
7	(Reserved)	Undefined	Undefined
8	BadVAddr	Bad virtual address	Exception
9	Count	Timer compare	Exception
10	EntryHi	High half of TLB entry (Virtual page number and ASID)	MMU
11	Compare	Timer compare	Exception
12	Status	Processor Status Register	Exception
13	Cause	Cause of the last exception taken	Exception
14	EPC	Exception Program Counter	Exception
15	PRId	Processor Revision Identifier	MMU
16	Config	Configuration Register	MMU
17	(Reserved)	Undefined	Undefined
18	(Reserved)	Undefined	Undefined
19	(Reserved)	Undefined	Undefined
20	(Reserved)	Undefined	Undefined
21	(Reserved)	Undefined	Undefined
22	(Reserved)	Undefined	Undefined
23	BadPAddr	Bad physical address	Exception
24	Debug	This is used for Debug function	Debug
25	Perf	Performance Counter and Control Register	Exception
26	(Reserved)	Undefined	Undefined
27	(Reserved)	Undefined	Undefined
28	TagLo	Cache Tag register (low bits)	Cache
29	TagHi	Cache Tag register (high bits)	Cache
30	ErrorEPC	Error Exception Program Counter	Exception
31	(Reserved)	Undefined	Undefined

4.2.1 Index Register (0)

31	30 … 6	5 … 0
P	**0**	**Index**
1	25	6

Figure 4-2. Index Register

The *Index* register is a 32-bit read/write register containing six bits to index an entry in the TLB. The high-order bit of the register records the success or failure of a *TLB Probe* (*TLBP*) instruction.

The *Index* register also specifies the TLB entry affected by *TLB Read* (*TLBR*) or *TLB Write Index* (*TLBWI*) instructions.

Table 4-2 shows the format of the *Index* register; Table 4-2 describes the *Index* register fields.

Table 4-2. Index Register Field Description

Field	Bits	Description	Type	Initial Value
P	31	Probe failure. Set to 1 when the previous TLB Probe (TLBP) instruction was unsuccessful.	Read/Write	Undefined
Index	5:0	Index to the TLB entry affected by the TLB Read and TLB Write instructions.	Read/Write	Undefined
0	30:6	Reserved. Must be written as zeroes, and returns zeroes when read.	Read-only	0

4.2.2 Random Register (1)

31 … 6	5 … 0
0	**Random**
26	6

Figure 4-3. Random Register

The *Random* register is a read-only register. The least significant six bits index an entry in the TLB. This register decrements every cycle an instruction is executed. Its value ranges between an upper and a lower bound, as follows:

- *A lower bound is set by the number of TLB entries reserved for exclusive use by the operating system (the contents of the* Wired *register).*
- *An upper bound is set by the total number of TLB entries (47 maximum).*

The *Random* register specifies the entry in the TLB that is affected by the *TLB Write Random* (TLBWR) instruction. The register does not need to be read for this purpose; however, the register is readable to verify proper operation of the processor.

To simplify testing, the *Random* register is set to the value of the upper bound upon system reset. This register is also set to the upper bound when the *Wired* register is written.

Figure 4-3 shows the format of the *Random* Register; Table 4-3 describes the *Random* Register fields.

Table 4-3. Random Register Fields

Field	Bits	Description	Type	Initial Value
Random	5:0	TLB Random index.	Read-only	Upper bound (47)
0	31:6	Reserved. Must be written as zeros, and returns zeroes when read.	Read-only	0

4.2.3 EntryLo0 Register (2), and EntryLo1 Register (3)

EntryLo0

31–26	25–6	5–3	2	1	0
0	**PFN**	**C**	**D**	**V**	**G**
6	20	3	1	1	1

EntryLo1

31–26	25–6	5–3	2	1	0
0	**PFN**	**C**	**D**	**V**	**G**
6	20	3	1	1	1

Figure 4-4. EntryLo0 and EntryLo1 Registers

The *EntryLo0* and *EntryLo1* registers consist of two registers that have similar format:

- *EntryLo0 is used for even virtual pages.*
- *EntryLo1 is used for odd virtual pages.*

The *EntryLo0* and *EntryLo1* registers are read/write registers. They hold the physical page frame number (PFN) of the TLB entry for even and odd pages, respectively, when performing TLB read and write operations.

Figure 4-4 shows the format of the *EntryLo0* and *EntryLo1* Registers; Table 4-4 describes the *EntryLo0* and *EntryLo1* Register fields.

Table 4-4. EntryLo0 and EntryLo1 Register Fields

Field	Bits	Description	Type	Initial Value
PFN	25:6	Page frame number; the upper bits of the physical address.	Read/Write	Undefined
C	5:3	Specifies the TLB page coherency attribute. 000(0): Reserved 001(1): Reserved 010(2): Uncached 011(3): Cacheable, write-back, write allocate 100(4): Reserved 101(5): Reserved 110(6): Reserved 111(7): Uncached Accelerated	Read/Write	Undefined
D	2	**Dirty.** If this bit is set, the page is marked as dirty and therefore writable. This bit is actually a write-protect bit that software can use to prevent alteration of data.	Read/Write	Undefined
V	1	**Valid.** If this bit is set, it indicates that the TLB entry is valid; otherwise, a TLBL or TLBS miss will occur.	Read/Write	Undefined
G	0	**Global.** If this bit is set in both EntryLo0 and EntryLo1, then the processor ignores the ASID during TLB look-up.	Read/Write	Undefined
0	31:26	Reserved. Must be written as zeroes, and returns zeroes when read. EntryLo0[31] is reserved for Kernel use. It contains the written value. This bit has no effect on any CPU or TLB operation.	Read-only	0

Reserved codes in C field may not be written correctly into TLB entry by TLBWI or TLBWR instruction.

4.2.4 Context Register (4)

31 — 23	22 — 4	3 — 0
PTEBase	**BadVPN2**	**0**
9	19	4

Figure 4-5. Context Register Format

The *Context* register is a read/write register containing the pointer to an entry in the page table entry (PTE) array. This array is an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the CPU loads the TLB with the missing translation from the PTE array. Normally, the operating system uses the *Context* register to address the current page map which resides in the kernel-mapped segment, kseg3. The *Context* register duplicates some of the information provided in the *BadVAddr* register, but the information is arranged in a form that is more useful for a software TLB exception handler. Figure 4-5 shows the format of the *Context* register; Table 4-5 describes the *Context* register fields.

Table 4-5. Context Register Fields

Field	Bits	Description	Type	Initial Value
PTEBase	31:23	This field is a read/write field for use by the operating system. It is normally written with a value that allows the operating system to use the Context register as a pointer into the current PTE array in memory.	Read/Write	Undefined
BadVPN2	22:4	This field is written by hardware on a miss. It contains the virtual page number (VPN) of the most recent virtual address that did not have a valid translation.	Read-only	Undefined
0	3:0	Reserved. Must be written as zeros, and returns zeroes when read.	Read-only	0

The 19-bit BadVPN2 field contains bits 31:13 of the virtual address that caused the TLB miss; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4 KB page size, this format can directly address the pair-table of 8-byte PTEs. For other page and PTE sizes, shifting and masking this value produces the appropriate address.

4.2.5 PageMask Register (5)

31 ... 25	24 ... 13	12 ... 0
0	**MASK**	**0**
7	12	13

Figure 4-6. PageMask Register

The *PageMask* register is a read/write register used for reading or writing the TLB. It holds a comparison mask that sets the variable page size for each TLB entry, as shown in Table 4-6.

Table 4-6. PageMask Register Field

Field	Bits	Description	Type	Initial Value
MASK	24:13	Page comparison mask. 0000 0000 0000: Page Size = 4 Kbytes 0000 0000 0011: Page Size = 16 Kbytes 0000 0000 1111: Page Size = 64 Kbytes 0000 0011 1111: Page Size = 256 Kbytes 0000 1111 1111: Page Size = 1 Mbytes 0011 1111 1111: Page Size = 4 Mbytes 1111 1111 1111: Page Size = 16 Mbytes	Read/Write	Undefined
0	31:25, 12:0	Reserved. Must be written as zeros, and returns zeroes when read.	Read-only	0

TLB read and write operations use this register as either a source or a destination; when virtual addresses are presented for translation into physical address, the corresponding bits in the TLB identify which virtual address bits among bits 24:13 are used in the comparison. When the Mask field is not one of the values shown in Table 4-6, the operation of the TLB is undefined.

4.2.6 Wired Register (6)

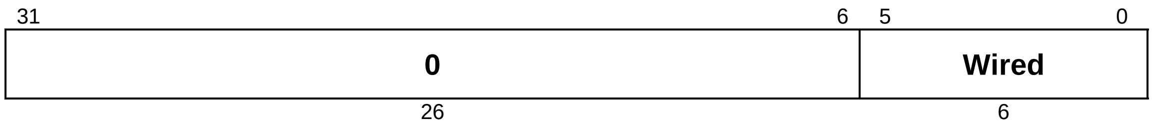


Figure 4-7. Wired Register

The *Wired* register is a read/write register that specifies the boundary between the wired and random entries of the TLB as shown in Figure 4-8. Wired entries are fixed, non-replaceable entries which cannot be overwritten by a TLB write operation. Random entries can be overwritten. Figure 4-7 shows the format of the *Wired* register. Table 4-7 describes the register fields.

The *Wired* register is set to 0 upon system reset. Writing this register also sets the *Random* register to the value of its upper bound as shown in Figure 4-8.

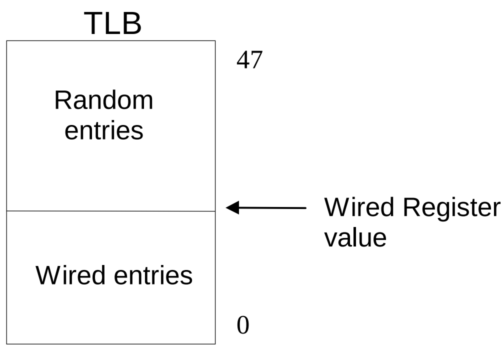


Figure 4-8. Wired Register Boundary

Writing a value greater than 47 into this register produces undefined results.

Table 4-7. Wired Register Field Descriptions

Field	Bits	Description	Type	Initial Value
Wired	5:0	TLB Wired boundary (the number of wired TLB entries)	Read/Write	0
0	31:6	Reserved. Must be written as zeros, and returns zeroes when read.	Read-only	0

4.2.7 BadVAddr Register (8)

31 … 0
BadVAddr
32

Figure 4-9. BadVAddr Register

The *Bad Virtual Address* register (*BadVAddr*) is a read-only register that displays the most recent virtual address that caused one of the following exceptions: TLB Invalid, TLB Modified, TLB Refill, or Address Error exceptions.

Figure 4-9 shows the format of the *BadVAddr* register; Table 4-8 describes the register fields.

Table 4-8. BadVAddr Register Field

Field	Bits	Description	Type	Initial Value
BadVAddr	31:0	The most recent virtual address that cause a TLB Invalid, TLB modified, TLB Refill, or Address Error exception.	Read-only	Undefined

Note: The *BadVAddr* register does not save any information for bus errors, since bus errors are not addressing errors.

4.2.8 Count Register (9)

31 ... 0
Count
32

Figure 4-10. Count Register

The *Count* register acts as a real-time timer. It is incremented every CPU clock cycle. The timer interrupt signaled through *IP[7]* can be disabled through the interrupt mask bit, *IM[7]*. This register can be read or written.

Figure 4-10 shows the format of the *Count* register. Table 4-9 describes the register fields.

Table 4-9. Count Register Field

Field	Bits	Description	Type	Initial Value
Count	31:0	32-bit timer, incrementing at the CPU clock rate.	Read/Write	Undefined

4.2.9 EntryHi Register (10)

31 … 13	12 … 8	7 … 0
VPN2	**0**	**ASID**
19	5	8

Figure 4-11. EntryHi Register

The *EntryHi* register holds the high-order bits of a TLB entry for TLB read and write operations. The *EntryHi* register is accessed by the *TLB Probe*, *TLB Write Random*, *TLB Write Indexed*, and *TLB Read Indexed* instructions.

When either a TLB Refill, TLB Invalid, or TLB Modified exception occurs, the *EntryHi* register is loaded with the virtual page number (VPN2) and the ASID of the virtual address that did not have a matching TLB entry.

Figure 4-11 shows the format of the *EntryHi* register. Table 4-10 describes the register fields.

Table 4-10. EntryHi Register Fields

Field	Bits	Description	Type	Initial Value
VPN2	31:13	Virtual page number divided by two (maps to two pages).	Read/Write	Undefined
ASID	7:0	Address space ID field. An 8-bit field that lets multiple processes share the TLB; each process can have a distinct mapping of otherwise identical virtual page numbers.	Read/Write	Undefined
0	12:8	Reserved. Must be written as zeroes, and returns zeroes when read.	Read-only	0

4.2.10 Compare Register (11)

31	0
Compare	

32

Figure 4-12. Compare Register

The *Compare* register acts as a timer (see also the *Count* register); it maintains a stable value that does not change on its own. When the value of the *Count* register equals the value of the *Compare* register, interrupt bit IP[7] in the *Cause* register is set. This causes an interrupt as soon as the interrupt is enabled. Writing a value to the *Compare* register, as a side effect, clears the timer interrupt.

For diagnostic purposes, the *Compare* register is a read/write register. In normal use, however, the *Compare* register is write-only. Figure 4-12 shows the format of the *Compare* register. Table 4-11 describes the register fields.

Table 4-11. Compare Register Field

Field	Bits	Description	Type	Initial Value
Compare	31:0	The Compare register saves a stable value compared to the Count register. When the value of the Count register equals to the value of the Compare register, interrupt IP[7] occurs.	Read/Write	Undefined

4.2.11 Status Register (12)

31 28	27	26	25 24	23	22	21 19	18	17	16	15	14 13	12	11 10	9 5	4 3	2	1	0
CU (CU[3:0])	**0**	**FR**	**0**	**DEV**	**BEV**	**0**	**CH**	**EDI**	**EIE**	**IM [7]**	**0**	**BEM**	**IM [3:2]**	**0**	**KSU**	**ERL**	**EXL**	**IE**
4	1	1	2	1	1	3	1	1	1	1	2	1	2	5	2	1	1	1

Figure 4-13. Status Register

The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. Figure 4-13 shows the format of the *Status* register. The following paragraphs identify the more important *Status* register fields and describe the fields. Some of the important fields include:

- *The 3-bit* Interrupt Mask (IM) *field controls the enabling of three interrupt signals. Interrupts must be enabled before they can be asserted. Interrupts are recognized by the processor when the corresponding bits are set in both the* Interrupt Mask *and the* Interrupt Enable *fields of the* Status *register and the* Interrupt Pending *field of the* Cause *register. The C790 does* not *support software interrupts.* IM[7] *corresponds to the internal timer interrupt and* IM[3:2] *corresponds to* **Int[1:0]** *signals.*
- *The 4-bit* Coprocessor Usability (CU) *field* (CU[3:0]) *controls the usability of four possible coprocessors. Regardless of the* CU[0] *bit setting, COP0 is always usable in Kernel mode. For all other cases, an access to an unusable coprocessor causes an exception. C790 supports coprocessor 1 (FPU).*

4.2.11.1 Status Register Format

Table 4-12 describes the *Status* register fields. All bits in the *Status* register are readable and writable.

Table 4-12. Status Register Fields

Field	Bits	Description	Type	Initial Value
CU (CU[3:0])	31:28	Controls the usability of each of the four coprocessor unit numbers. COP0 is always usable when in Kernel mode, regardless of the setting of the CU[0] bit. 1 → usable 0 → unusable	Read/ Write	Undefined
FR	26	Enable additional floating point registers 0 → 16 registers 1 → 32 registers	Read/ Write	0
DEV	23	Controls the location of Performance counter and debug/SIO exception vectors. 0 → normal 1 → bootstrap	Read/ Write	Undefined
BEV	22	Controls the location of TLB refill and general exception vectors. 0 → normal 1 → bootstrap	Read/ Write	1
CH	18	Cache Hit (tag match and valid state) or Miss indication for last CACHE Hit Invalidate and CACHE Hit Write-back Invalidate for the Data cache. 0 → miss 1 → hit	Read/ Write	Undefined
EDI	17	***EI/DI instruction Enable:*** When this bit is set, the EI and DI instructions can operate in User, Supervisor and Kernel modes and as such set or clear the *EIE* bit to enable or disable all interrupts (except NMI). When this bit is cleared, EI and DI operate as NOPs in User and Supervisor modes and executes properly in Kernel mode.	Read/ Write	Undefined
EIE	16	***Enable IE***: This bit enables or disables the IE (Interrupt Enable) bit. This bit is cleared by the DI instruction and set by the EI instruction. 0 → disables all interrupts regardless of the value of the *IE* bit. 1 → enables the *IE* bit. (All interrupts are enabled if *IE*=1, *EXL*=0, and *ERL*=0.) Note: IM enables individual interrupt	Read/ Write	Undefined
IM[7,3:2]	15, 11:10	***Interrupt Mask*:** controls the enabling of each of the external and internal interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bits are set in both the Interrupt Mask field of the Status register and the Interrupt Pending field of the Cause register. 0 → disabled 1 → enabled Note: The enabling of this bit is valid only when EIE=1, IE=1, EXL=0 and ERL=0	Read/ Write	Undefined
BEM	12	***Bus Error Mask:*** controls the updating of the BadPAddr register and signaling a bus error exception. 0 → update BadPAddr and signal a bus error exception. 1 → do not update BadPAddr and stop signaling a bus error exception. This bit is set to 1 when it is a 0 and a bus error is signaled.	Read/ Write	Undefined
KSU	4:3	***Kernel/Supervisor/User Mode bits:*** 00_2 → Kernel 01_2 → Supervisor 10_2 → User 11_2 → Reserved	Read/ Write	Undefined

Field	Bits	Description	Type	Initial Value
ERL	2	***Error Level:*** set by the processor when Reset, NMI, performance counter, SIO or debug exception is taken. 0 → normal 1 → error	Read/ Write	1
EXL	1	***Exception Level:*** set by the processor when any exception other than Reset, NMI, performance counter, or debug exception is taken. 0 → normal 1 → exception	Read/ Write	Undefined
IE	0	***Interrupt Enable*** 0 → disables all interrupts 1 → enables all interrupts (if EIE=1, ERL=0, and EXL=0)	Read/ Write	Undefined
0	27, 25:24, 21:19, 14:13, 9:5	Reserved. Must be written as zeroes, and returns zeroes when read.	Read-only	0

4.2.11.2 Status Register Modes and Access States

Fields of the *Status* register set the modes and access states below.

Interrupt Enable: Interrupts are enabled when all of the following conditions are true:

- Status.IE *= 1,*
- and Status.EIE *= 1,*
- *and* Status.EXL *= 0,*
- *and* Status.ERL *= 0*

If these conditions are met, setting the *IM* bits enable the appropriate interrupts.

SIO Enable: A level 2 exception by SIO is enabled when the following condition is true:

- Status.ERL *= 0*

If this condition is met, asserting the **SIO** signal causes a Debug exception to occur.

Operating Modes: The following CPU *Status* register bit settings are required for *User*, *Kernel*, and *Supervisor* modes.

- *The Processor is in* User *mode when* KSU *=* 10_2 *and* EXL *= 0 and* ERL *= 0.*
- *The processor is in* Supervisor *mode when* KSU *=* 01_2 *and* EXL *= 0 and* ERL *= 0.*
- *The processor is in* Kernel *mode when* KSU *=* 00_2 *or* EXL *= 1 or* ERL *= 1.*

Kernel Address Space Accesses: Access to the kernel address space is allowed when the processor is in Kernel mode.

Supervisor Address Space Accesses: Access to the supervisor address space is allowed when the processor is in Kernel mode or Supervisor mode, as described above.

User Address Space Accesses: Access to the user address space is allowed in Kernel, Supervisor, and User modes.

4.2.12 Cause Register (13)

31	30	29 28	27 19	18 16	15	14 13	12	11 10	9 7	6 2	1 0
BD	BD2	CE	0	EXC2	IP [7]	0	SIOP	IP [3:2]	0	ExcCode	0
1	1	2	9	3	1	2	1	2	3	5	2

Figure 4-14. Cause Register

The 32-bit read-only *Cause* register describes the cause of the most recent exception. Figure 4-14 shows the fields of this register. Table 4-13 describes the *Cause* register fields. All bits in the *Cause* register are read-only.

Table 4-13. Cause Register Fields

Field	Bits	Description	Type	Initial Value
BD	31	Set by the processor when any exception other than Reset, NMI, performance counter, or debug occurs and is taken in a branch delay slot. 1 → delay slot 0 → normal	Read-only	Undefined
BD2	30	Indicates whether the last NMI, performance counter, debug, or SIO exception taken occurred in a branch delay slot. 1 → delay slot 0 → normal	Read-only	Undefined
CE	29:28	Coprocessor unit number referenced when a Coprocessor Unusable exception is taken.	Read-only	Undefined
EXC2	18:16	Indicates the exception codes for level 2 exceptions (Performance Counter, Reset, Debug, SIO and NMI exceptions) 000 (0) : Res (Reset) 001 (1) : NMI (Non-maskable Interrupt) 010 (2) : PerfC (Performance Counter) 011 (3) : Dbg (Debug) and SIO (SIO) 1xx (4-7) : Reserved	Read-only	Undefined
IP[7,3:2]	15, 11:10	Indicates an interrupt is pending. 1 → interrupt pending 0 → no interrupt	Read-only	Undefined, ***Int[1:0]***
SIOP	12	Indicates an SIO signal is pending 1 → SIO signal is pending 0 → no SIO signal is pending	Read-only	***SIO***

Field	Bits	Description	Type	Initial Value
ExcCode	6:2	Exception code filed. 00000 (0) : Int (Interrupt) 00001 (1) : Mod (TLB modification exception) 00010 (2) : TLBL (TLB exception (load or instruction fetch)) 00011 (3) : TLBS (TLB exception (store)) 00100 (4) : AdEL (Address error exception (load or instruction fetch)) 00101 (5) : AdES (Address error exception (store)) 00110 (6) : IBE (Bus error exception (instruction fetch)) 00111 (7) : DBE (Bus error exception (data reference: load or store)) 01000 (8) : Sys (Syscall exception) 01001 (9) : Bp (Breakpoint exception) 01010 (10): RI (Reserved instruction exception) 01011 (11): CpU(Coprocessor Unusable exception) 01100 (12): Ov (Arithmetic overflow exception) 01101 (13): Tr (Trap exception) 01110 (14): Reserved 01111 (15): FPE Floating-Point exception (16-31): (Reserved)	Read-only	Undefined
0	27:19, 14:13, 9:7, 1:0	Reserved. Must be written as zeroes, and returns zeroes when read.	Read-only	0

4.2.13 EPC Register (14)

31 ... 0
EPC
32

Figure 4-15. EPC Register

The *Exception Program Counter (EPC)* is a read/write register that contains the address at which processing resumes after an exception has been serviced.

For synchronous exceptions, the *EPC* register contains either:

- *the virtual address of the instruction that was the direct cause of the exception, or*
- *the virtual address of the immediately preceding branch or jump instruction (when the instruction is in a branch delay slot, and the* BD *bit in the* Cause *register is set).*

On the occurrence of an exception, if the *EXL* bit in the *Status* register is set to a 1, the processor does not update the *EPC* register. Figure 4-15 shows the format of the *EPC* register. Table 4-14 describes the *EPC* register fields.

Table 4-14. EPC Register Field

Field	Bits	Description	Type	Initial Value
EPC	31:0	Contains the address at which processing can resume after an exception has been serviced.	Read/Write	Undefined

4.2.14 PRId Register (15)

31 16	15 8	7 0
0	**Imp**	**Rev**
16	8	8

Figure 4-16. PRId Register

The 32-bit read-only *Processor Revision Identifier (PRId)* register contains information identifying the implementation and revision level of the C790 and COP0. Figure 4-16 shows the format of the *PRId* register; Table 4-15 describes the *PRId* register fields.

The low-order byte (bits 7:0) of the *PRId* register is interpreted as a revision number, and the high-order byte (bits 15:8) is interpreted as an implementation number. The implementation number of the C790 processor is **0x38**. The content of the high-order halfword (bits 31:16) of the register are reserved.

The revision number is stored as a value in the form *y.x,* where *y* is major revision number in bits 7:4 and *x* is a minor revision number in bits 3:0.

The revision number can distinguish some chip revisions, but there is no guarantee that changes to the chip will necessarily be reflected in the *PRId* register, or that changes to the revision number necessarily reflect real chip changes. For this reason, these values are not listed and software should not rely on the revision number in the *PRId* register to characterize the chip.

Table 4-15. PRId Register Fields

Field	Bits	Description	Type	Initial Value
Imp	15:8	Implementation number	Read-only	0x38
Rev	7:0	Revision number of each mask	Read-only	Revision number
0	31:16	Reserved. Must be written as zeroes, and returns zeroes when read.	Read-only	

4.2.15 Config Register (16)

31	30 28	27 19	18	17	16	15	14	13	12	11 9	8 6	5 3	2 0
0	EC	0	DIE	ICE	DCE	BE	0	NBE	BPE	IC	DC	0	K0
1	3	9	1	1	1	1	1	1	1	3	3	3	3

Figure 4-17. Config Register Format

The *Config* register specifies various configuration options which can be selected. Figure 4-17 shows the format of the *Config* register; Table 4-16 describes the *Config* register fields.

Some configuration options, as defined by *Config* bits 30:28, 15 and 11:6, are set by the hardware during reset and are included in the *Config* register as read-only status bits for the software to access. Other configuration options like 18:16 and 13:12 are set by hardware during reset and can be modified by software. Other configuration options like bits 2:0 are read/write and controlled by software; on reset these fields are undefined.

Table 4-16. Config Register Fields

Field	Bits	Description	Type	Initial Value
EC	30:28	Bus clock ratio. 000: processor clock frequency divided by 2 001 ~ 111: (Reserved)	Read-only	0
DIE	18	Double issue enable 0 → Single issue 1 → Double issue	Read/Write	0
ICE	17	Setting this bit to 1 enables the instruction cache. 0 → Instruction cache disable 1 → Instruction cache enable The CACHE instruction for the instruction cache is enabled regardless of the value of this bit.	Read/Write	0
DCE	16	Setting this bit to 1 enables the data cache. 0 → Data cache disable 1 → Data cache enable If the cache is disabled, the PREF instruction becomes a NOP.	Read/Write	0
BE	15	Big Edian 0 → Little Edian 1 → Big Edian	Read-only	Pin
NBE	13	Setting this bit to 1 enables non-blocking load. 0 → Disable Non-blocking loads and hit under miss 1 → Enable Non-blocking loads and hit under miss	Read/Write	0
BPE	12	Setting this bit to 1 enables branch prediction. 0 → Disable Branch Prediction 1 → Enable Branch Prediction	Read/Write	0
IC	11:9	Instruction cache Size (Instruction cache size = 2^{12+IC} bytes). 011 → 32 KB	Read-only	011
DC	8:6	Data cache Size (Data cache size = 2^{12+DC} bytes). 011 → 32 KB	Read-only	011

Field	Bits	Description	Type	Initial Value
K0	2:0	*kseg0* coherency algorithm. 000: Reserved 001: Reserved 010: Uncached 011: Cacheable, write-back, write allocate 100: Reserved 101: Reserved 110: Reserved 111: Uncached Accelerated	Read/Write	Undefined
0	31, 27:19, 14, 5:3	Reserved, Must be written as zeroes, and returns zeroes when read.	Read-only	0

With single issue enabled (DIE = 0), the C790 always fetches two instructions but only issues a single instruction.

4.2.16 BadPAddr Register (23)

31 … 4	3 … 0
BdPAddr	**0**
28	4

Figure 4-18. BadPAddr Register Format

The *Bad Physical Address* register (*BadPAddr*) is a read-only register that contains the most recent physical address that caused a bus error. It is updated with a new value whenever *Status.BEM* is clear (0). Once this bit is set (on the occurrence of a bus error) the register holds the value.

Figure 4-18 shows *BadPAddr* register format; Table 4-17 describes the register fields.

Table 4-17. BadPAddr Register Fields

Field	Bits	Description	Type	Initial Value
BdPAddr	31:4	Physical Address value	Read-Only	undefined
0	3:0	Reserved. Returns zeros when read.	Read-Only	0

4.2.17 Debug Registers (24)

There are seven separately addressable debug registers, which are all assigned to CP0, register 24.

Each of the seven registers is accessed by specifying subaccess code which is bit2 to bit0 of an instruction code.

Breakpoint Control Register (BPC) (subaccess code 0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14 3	2	1	0
IAE	DRE	DWE	DVE	0	IUE	ISE	IKE	IE	0	DUE	DSE	DKE	DXE	ITE	DTE	BED	0	DWB	DRB	IAB

See Table 13-3 for a detailed description of individual BPC register fields.

Instruction Address Breakpoint (IAB) (subaccess code 2)

31 ... 2	1 ... 0
IAB	0
30	2

Instruction Address Breakpoint Mask Register (IABM) (subaccess code 3)

31 ... 2	1 ... 0
IABM	0
30	2

Data Address Breakpoint Register (DAB) (subaccess code 4)

31 ... 0
DAB
32

Data Address Breakpoint Mask Register (DABM) (subaccess code 5)

31 ... 0
DABM
32

Data value Breakpoint Register (DVB) (subaccess code 6)

31 ... 0
DVB
32

Data value Breakpoint Mask Register (DVBM) (subaccess code 7)

31 ... 0
DVBM
32

4.2.18 Performance Counter Registers (25)

There are three separately addressable performance counter registers, which are all assigned to COP0, register 25.
Each of the three registers is accessed by specifying subaccess code which is bit1 to bit0 of an instruction code.
All performance counter registers are read/write registers.

Performance Counter Control Register (PCCR)

31	30–20	19–15	14	13	12	11	10	9–5	4	3	2	1	0
CTE	0	EVENT1	U1	S1	K1	EXL1	0	EVENT0	U0	S0	K0	EXL0	0
1	11	5	1	1	1	1	1	5	1	1	1	1	1

Performance Counter Register 0 (PCR0)

31	30–0
OVFL	VALUE
1	31

Performance Counter Register 1 (PCR1)

31	30–0
OVFL	VALUE
1	31

Figure 4-19. Performance Counter Registers

Table 4-18 lists the field definitions for the *Performance Counter Control* register.

Table 4-18. Performance Counter Control Register Fields

Field	Bits	Description	Type	Initial Value
CTE	31	Enables event counting (CTR1, CTR0) and exception generation: 0 → Disable 1 → Enable	Read/Write	0
EVENT1	19:15	Set the event to be monitored by PCR1 00000 (0) Low-order branch issued 00001 (1) Processor cycle 00010 (2) Dual instruction issue 00011 (3) Branch miss predicted 00100 (4) TLB miss 00101 (5) DTLB miss 00110 (6) Data Cache miss 00111 (7) WBB single request unavailable 01000 (8) WBB burst request unavailable 01001 (9) WBB burst request almost full 01010 (10) WBB burst request full 01011 (11) CPU data bus busy 01100 (12) Instruction completed 01101 (13) Non-BDS instruction completed 01110 (14) COP1 instruction completed 01111 (15) Store completed 10000 (16) No event (17-31) Reserved	Read/Write	Undefined
EVENT0	9:5	Set the event to be monitored by PCR0 00000 (0) Reserved 00001 (1) Processor cycle 00010 (2) Single instruction issue 00011 (3) Branch issue 00100 (4) BTAC miss 00101 (5) ITLB miss 00110 (6) Instruction Cache miss 00111 (7) DTLB accessed 01000 (8) Non-blocking load 01001 (9) WBB single request 01010 (10) WBB burst request 01011 (11) CPU address bus busy 01100 (12) Instruction completed 01101 (13) Non-BDS instruction completed 01110 (14) Reserved 01111 (15) Load completed 10000 (16) No event (17-31) Reserved.	Read/Write	Undefined
U1, U0	14, 4	Enables event counting (PCR1/PCR0) in the User mode. 0 → Disable 1 → Enable	Read/Write	Undefined
S1, S0	13, 3	Enables event counting (PCR1/PCR0) in the Supervisor mode. 0 → Disable 1 → Enable	Read/Write	Undefined
K1, K0	12, 2	Enables event counting (PCR1/PCR0) in the Kernel mode. 0 → Disable 1 → Enable	Read/Write	Undefined
EXL1, EXL0	11, 1	Enables event counting (PCR1/PCR0) when EXL bit is set in the *Status* register. 0 → Disable 1 → Enable	Read/Write	Undefined
0	30:20, 10, 0	Reserved. Must be written as zero, and returns zero when read.	Read-only	0

Table 4-19 lists the field definitions for the *Performance Counter* register *0* (*PCR0*).

Table 4-19. Performance Counter Register 0 Fields

Field	Bits	Description	Type	Initial Value
OVFL	31	Overflow flag	Read/Write	Undefined
VALUE	30:0	The actual counter	Read/Write	Undefined

Table 4-20 lists the field definitions for the *Performance Counter* register *1* (*PCR1*).

Table 4-20. Performance Counter Register 1 Fields

Field	Bits	Description	Type	Initial Value
OVFL	31	Overflow flag	Read/Write	Undefined
VALUE	30:0	The actual counter	Read/Write	Undefined

4.2.19 TagLo (28) and TagHi (29) Registers

TagLo

31–12	11–7	6	5	4	3	2–0
PTagLo	**Special use**	**D**	**V**	**R**	**L**	**Su**
20	5	1	1	1	1	3

TagHi

31–0
Special use
32

Figure 4-20. TagLo and TagHi Registers

The *TagLo* and *TagHi* registers are 32-bit read/write registers used by the CACHE instruction. For writing to the data cache tags, the *TagLo* register contains the fields as shown above and the *TagHi* register is not used. For writing to the data cache data portion the *TagLo* register contains the data value. For writing to the instruction cache tags the *TagLo* register contains the fields as defined above except that bits three and six are also reserved bits. For writing to the instruction cache data portion, the *TagLo* register contains the data (instruction) and the *TagHi* register contains the steering bits and bits for the BHT as defined in Chapter 7. When reading from the caches, the values in the *TagLo* and *TagHi* register are the same as described above for writing. These registers are also used for manipulating the BTAC. See the description of the CACHE instruction in Appendix C for details. Figure 4-20 shows the format of these registers for some of the cache operations.

Table 4-21 lists the field definitions of the *TagLo* register.

Table 4-21. TagLo Register Fields

Field	Bits	Description	Type	Initial Value
PTagLo [31:12]	31:12	PTagLo[31:12] specifies 20-bit physical address tag cache.	Read/Write	Undefined
D	6	**Dirty:** 0 → Clean 1 → Dirty	Read/Write	Undefined
V	5	**Valid:** 0 → Invalid 1 → Valid	Read/Write	Undefined
R	4	**LRF Replacement:** This bit participates in the calculation determining which cache way will be used for the next replacement. See Section 7.3.1 for details.	Read/Write	Undefined
L	3	**Lock:** This bit is *only* used for the data cache. For instruction cache operations this bit is treated as a reserved bit. 0 → For this line, this side is not locked. 1 → For this line, this side is locked.	Read/Write	Undefined
Special use, Su	11:7, 2:0	Used by the CACHE instruction to manipulate the branch target address cache. Refer to Chapter 7 for details.	Read/Write	Undefined

Table 4-22. TagHi Register Fields

Field	Bits	Description	Type	Initial Value
Special use	31:0	The TagHi register is used by the CACHE instruction to manipulate some of the bits of the instruction cache. Refer to Chapter 7 for details.	Read/Write	Undefined

4.2.20 ErrorEPC (30)

31 ... 0
ErrorEPC
32

Figure 4-21. ErrorEPC Register

The *ErrorEPC* register is similar to the *EPC* register, except that *ErrorEPC* is used on nonmaskable interrupt (NMI), debug, SIO, and performance counter exceptions.

The read/write *ErrorEPC* register contains the virtual address at which instruction processing can resume after servicing an error. This address can be:

- *the virtual address of the instruction that caused the exception*
- *the virtual address of the immediately preceding branch or jump instruction (when the instruction is in a branch delay slot, and the* BD2 *bit in the* Cause *register is set).*

Table 4-23 lists the field definition of the *ErrorEPC* register.

Table 4-23. ErrorEPC Register Field

Field	Bits	Description	Type	Initial Value
ErrorEPC	31:0	Contains the virtual address at which instruction processing can resume after servicing an error.	Read/Write	Undefined

5. Exception Processing and Reset

This chapter describes the exception processing, including level 1 and level 2 exceptions.

5.1 The Exception Handling Process

Exceptions can be recognized while the program is any of its three operating modes: User, Supervisor, or Kernel.

Exceptions are categorized into 2 groups which are level 1 exceptions and level 2 exceptions as shown in Table 5-1.

Table 5-1. Exception Levels

Level 1 Exceptions	Level 2 Exceptions
Interrupt	Reset
TLB Modified	NMI
TLB Refill	Performance Counter
TLB Invalid	Debug
Address Error	SIO
Syscall	
Break	
Trap	
Reserved Instruction	
Coprocessor Unusable	
Integer Overflow	
Bus Error	
Floating Point Exception	

Compatibility Note: Level 2 exceptions are a generalization of “error level” exception processing defined in earlier MIPS implementation.

5.1.1 Level 1 Exceptions

Exception Processing

When the processor takes a level 1 exception, the processor switches to Kernel mode. Rather than set the *Status.KSU* bits to effect the switch, the *Status.EXL* bit is set to 1. Whenever *Status.EXL* is 1, the operating mode is Kernel mode, regardless of the setting of *Status.KSU*.

Then the processor saves the virtual address of the instruction canceled by the exception. This address is saved in the *EPC* register. If the canceled instruction is in the delay slot of a branch instruction, the *Cause.BD* bit is set to 1 and *EPC* is set to the address of the branch instruction (rather than the delay slot). For non-delay-slot instructions, *Cause.BD* is set to 0. If *Status.EXL* bit was 1 before the exception is taken, EPC and *Cause.BD* aren’t set. The exception service routine examines *Cause.BD* to determine the true address of the instruction that raised the exception.

In addition to setting *EPC*, *Cause.BD*, and *Status.EXL*, the 5 bit field *Cause.ExcCode* is also set. This field specifies the cause of the exception; The *Cause.CE* fields may also get set when an Coprocessor unusable exception is raised.

After setting those bits, the processor jumps to the exception vector address.

The basic exception handling operation performed can be described using the Figure 5-1 Level 1 Exception Processing Flowchart.

(see next page)

Disabled exceptions in level 1 exception handler

Once a level 1 exception service routine is entered, interrupts and bus error are unconditionally disabled.

> *C790 Programming Note:* The only level 1 exception that is unconditionally disabled within level 1 exceptions handler is external interrupts and bus errors. All other level 1 exceptions still occur and are recognized (if enabled). a software system that makes use of such exceptions must use extreme care. In particular, it must make sure that it has saved *EPC* and *Cause.BD* somewhere (e.g. in a software managed stack) before the exception occurs.

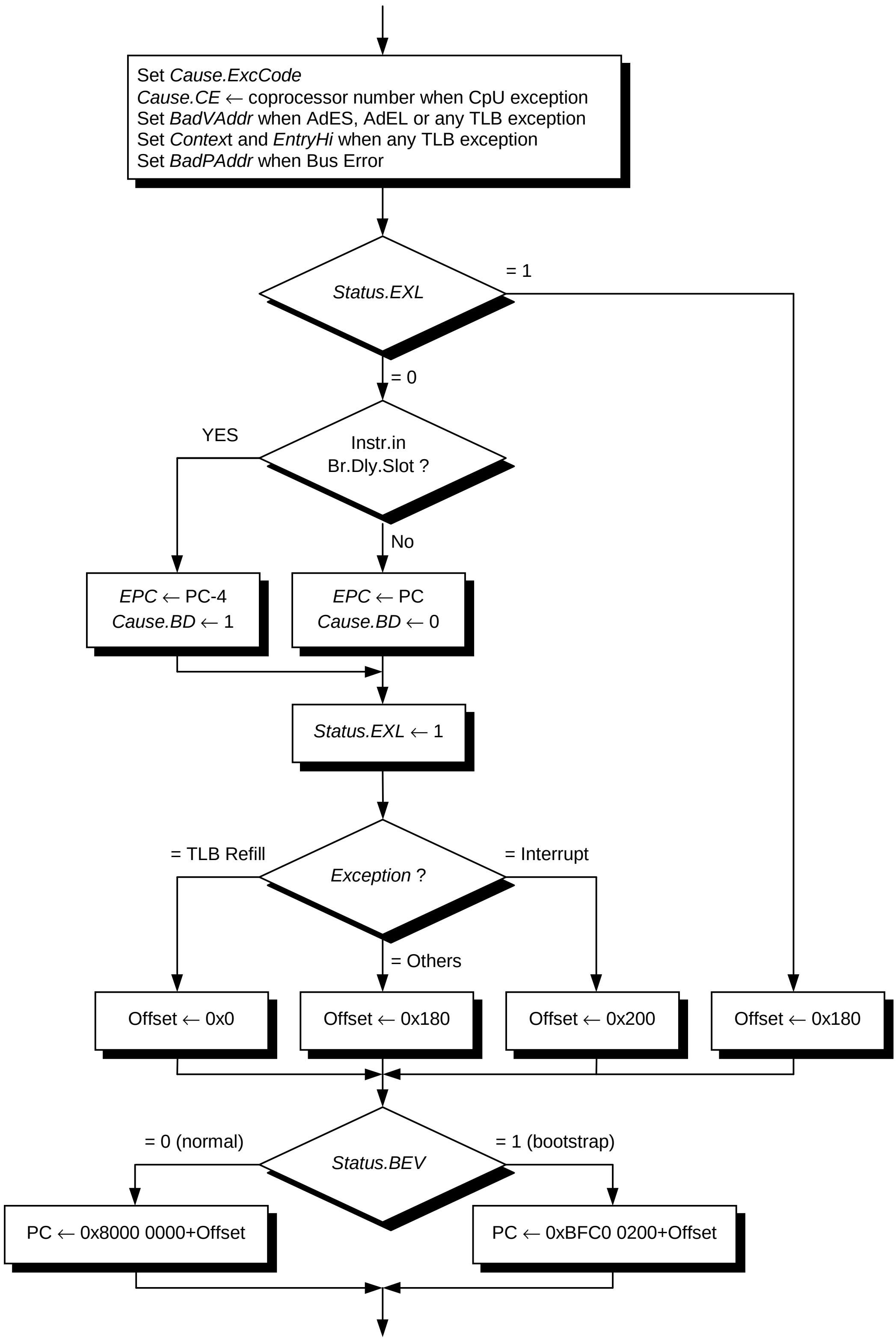


Figure 5-1. Level 1 Exception processing flowchart

5.1.2 Level 2 Exceptions

Exception Processing

When the processor takes a level 2 exception, the processor switches to kernel mode, by *setting Status.ERL* to 1.

The address of the instruction where the Level 2 exception was recognized is stored in the *ErrorEPC* register. If the canceled instruction is in the delay slot of a branch instruction, the *Cause.BD2* bit is set to 1 and *ErrorEPC* is set to the address of the branch instruction (rather than the delay slot). For non-delay-slot instructions, *Cause.BD2* is set to 0. In addition, the cause of the exception is stored in *Cause.EXC2*.

After setting those bits, the processor jumps to the exception vector address.

The basic Level 2 exception handling operation performed can be described using the Figure 5-2 Level 2 Exception processing Flowchart.

(see next page)

Disabled Exceptions in level 2 exceptions

When executing a Level 2 exception service routine, following exceptions are disabled.

- NMI, Interrupt, and Bus error
- Debug, SIO and Performance counter

C790 Implementation Note: Any external exception that is not level-sensitive (e.g. NMI) must be held until it is recognized; i.e. at least until the Level 2 handler is exited.

C790 Programming Note: It is the programmer's responsibility to ensure that all other internal exceptions (e.g. OVERFLOW) never occur within a Level 2 handler. If they do occur, the corresponding Level 1 exception handler will be entered. Since both *Status.EXL* and *Status.ERL* will be set when servicing this (nested) exception, the ERET used to exit the service routine will operate incorrectly.

C790 Programming Note: When *Status.ERL* = 1, the user address, *Kuseg*, region becomes a 2^{31}-byte unmapped, uncached address space (that is, mapped directly to physical address 0x0000 0000-0x7FFF FFFF).

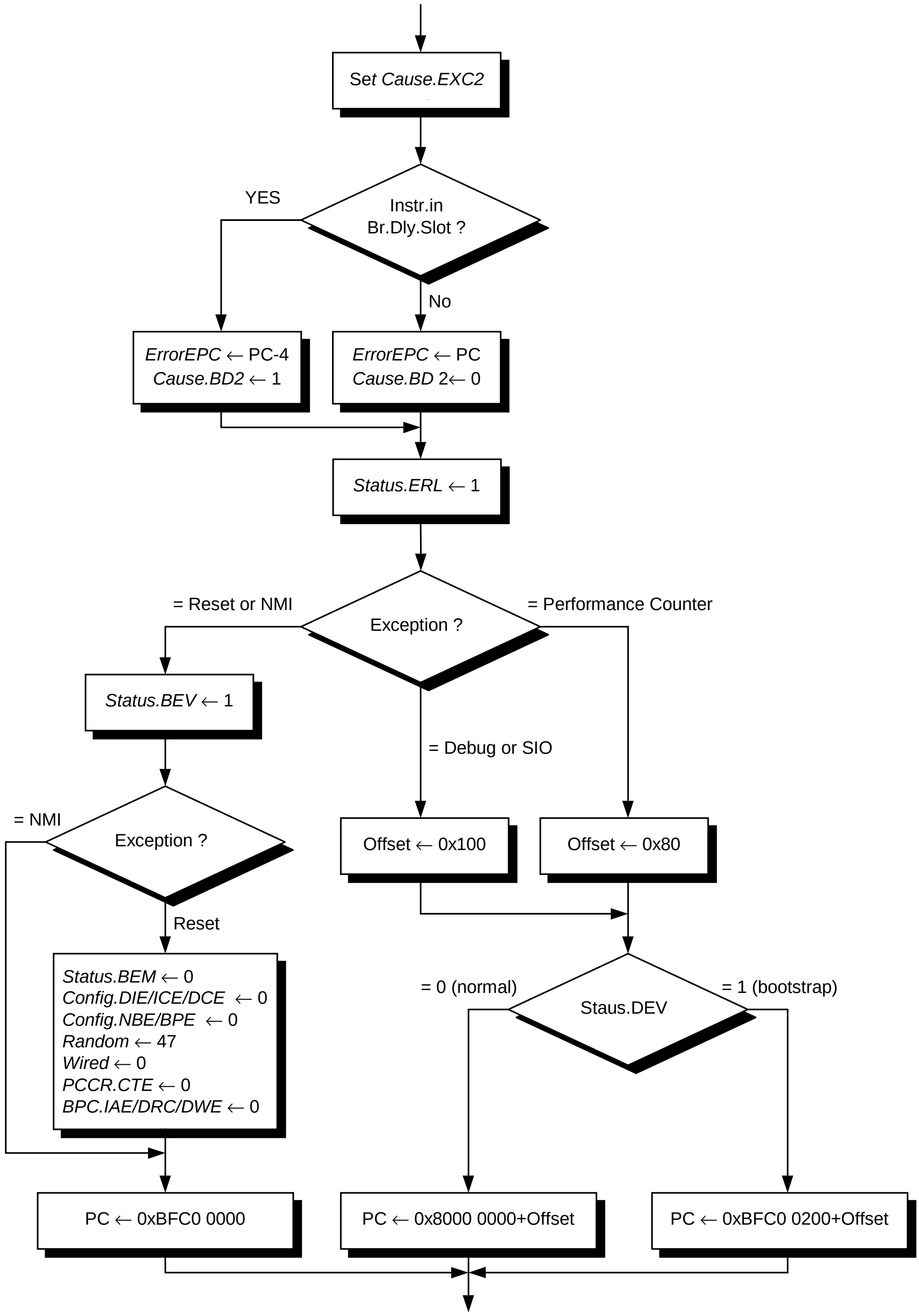


Figure 5-2. Level 2 Exception processing flowchart

5.2 Exception Vector Locations

Exception vector addresses for level 1 exceptions are shown in Table 5-2.
The vector address for TLB refill depends on the *Status.EXL* bit. The vector addresses for level 1 exceptions also depend on the *Status.BEV* bit.

Table 5-2. Exception Vectors for Level 1 exceptions

Exceptions	Vector Address	
	BEV = 0	**BEV = 1**
TLB Refill (EXL = 0)	0x8000 0000	0xBFC0 0200
TLB Refill (EXL = 1)	0x8000 0180	0xBFC0 0380
Interrupt	0x8000 0200	0xBFC0 0400
Others	0x8000 0180	0xBFC0 0380

Exception vector addresses for level 2 exceptions are shown in Table 5-3.
The vector addresses for level 2 exceptions also depend on the *Status.DEV* bit.

Table 5-3. Exception Vectors for Level 2 exceptions

Exceptions	Vector Address	
	DEV = 0	**DEV = 1**
Reset, NMI	0xBFC0 0000	0xBFC0 0000
Performance Counter	0x8000 0080	0xBFC0 0280
Debug, SIO	0x8000 0100	0xBFC0 0300

5.3 Cause Register Setting

The *Cause.ExcCode* bits are set when a level 1 exception is taken.
The *Cause.ExcCode* setting is shown in Table 5-4.

Table 5-4. Cause.ExcCode Field

ExcCode	Exception
0	Int (Interrupt)
1	Mod (TLB modification exception)
2	TLBL (TLB exception; load or inst fetch)
3	TLBS (TLB exception; store)
4	AdEL (Address error exception; load or inst fetch)
5	AdES (Address error exception; store)
6	IBE (Bus error exception; instruction fetch)
7	DBE (Bus error exception; load or store)
8	Sys (Syscall exception)
9	Bp (Breakpoint exception)
10	RI (Reserved instruction exception)
11	CpU (Coprocessor Unusable exeption)
12	Ov (Integer Overflow exception)
13	Tr (Trap exception)
14	Reserved
15	FPE (Floating Point Exception)
16-31	Reserved

The *Cause.EXC2* bits are set when a level 2 exception is taken.
The *Cause.EXC2* setting is shown in Table 5-5.

Table 5-5. Cause.EXC2 Field

EXC2	Exception
0	Res (Reset exception)
1	NMI (Non-Maskable Interrupt)
2	PerfC (Performance Counter exception)
3	Dbg (Debug exception), SIO (SIO exception)
4	SS (Single Step)
5-7	Reserved

5.4 Masking an exception

The following exceptions can be masked by setting bits in Status register.

NMI, Performance counter, Debug, Bus error, Interrupt and SIO

The Table 5-6 shows whether the bits mask those exceptions. Exceptions which marked with “X” can be masked by setting (BEM, EXL or ERL) or clearing (IE or IM) the corresponding bit in the Status register.

Table 5-6. Masking exceptions

Exception	Mask bit (in Status register)				
	IE	IM	BEM	EXL	ERL
Reset					
NMI					X
Performance Counter					X
Debug					X
SIO					X
Address error					
TLB Refill/Invalid/Modify					
Bus error			X	X	X
Syscall					
Break					
Reserved instrcution					
Coprocessor Unusable					
Interrupt	X	X		X	X
Integer overflow					
Trap					

5.5 Detaild Description

5.5.1 Exception Priority

Exception priority rules determine which exception is taken first, if multiple exceptions occur on the same instruction. The Table 5-7. Shows the priority order of the exceptions.

Table 5-7. Exception Priority Order

Reset ***(highest priority)***
NMI
Performance Counter
Instruction Breakpoint (debug)
Address error - Instruction fetch
TLB refill - Instruction fetch
TLB invalid - Instruction fetch
Bus Error - Instruction fetch
Single Step
SYSCALL, BREAK, Reserved Instruction,* Floating Point Exception or Coprocessor Unusable*
Interrupt
Data address/value breakpoint (debug)
SIO
Integer overflow, Trap
Address error - data access
TLB refill - data access
TLB invalid - data access
TLB modified - data access
Bus error - data access ***(lowest priority)***

* The exception priority between Reserved Instruction exception(RI) and Coprocessor Unusable exception(CpU)

The exception priorities of the two exceptions are the same. However, when Status.CU[1] = 0, an attempt to execute any FPU (COP1) instruction causes a CpU exception. When Status.CU[1] = 1, the attempt is reported as an FPE(E):unimplemented FPU exception in the Cop1 sub-instructions.
On the other hand, an attempt to execute any COP0 class Reserved Instruction causes an RI exception regardless Status.CU[0].

5.5.2 Reset Exception

Cause

The RESET exception occurs when the ***Reset**** signal is asserted and then deasserted. This exception is not maskable.

Exception Level: 2

Vector Address: 0xBFC00000

Processing

The RESET exception vector is located within uncached and unmapped address space. Hence the cache and TLB need not be initialized in order to process the exception.

The contents of all registers in the CPU are undefined when this exception is recognized, except for the following register fields:

- In the *Status* register,
  Status.*ERL* and Status.*BEV* are set to 1.
  Status.BEM is set to 0.
  All other bits except for 0-fixed bits are undefined.
- In the *Cause* register,
  Cause.*EXC2* is set to 0 (to indicate that a Reset occurred)
  All other bits except for 0-fixed bits are undefined.
- In the *Config* register,
  DIE, *ICE*, *DCE*, *NBE*, and *BPE* bits are set to 0.
  All other bits except for fixed-value, read-only bits are undefined.
- The *Random* register is initialized to the value of its upper bound (47).
- The *Wired* register is initialized to 0.
- The Counter Enable flag in the Performance Counter Control register (*PCCR.CTE*) is set to 0.
- The breakpoint address enable flags in the Breakpoint Control register, *BPC.IAE*, *BPC.DRE*, and *BPC.DWE,* are all set to 0.
- Valid, Dirty, LRF, and Lock bits of the data cache and the Valid and LRF bits of the instruction cache are initialized to 0 on reset.

Servicing

The RESET exception is serviced by:

- initializing all processor registers, coprocessor registers, caches, and the memory system
- performing diagnostic tests
- bootstrapping the operating system

5.5.3 Non-Maskable Interrupt (NMI) Exception

Cause

The Non-Maskable Interrupt (NMI) exception occurs in response to the falling edge of the ***NMI**** signal. The NMI exception is maskable by setting the *Status.ERL* bit. It is recognized regardless of the settings of the *Status.EXL,* and *Status.IE* bits.

Exception Level: 2

Vector Address: 0xBFC00000

Processing

NMI and RESET exceptions share the same exception vector. This vector is located within uncached and unmapped address space; therefore, the cache and TLB need not be initialized in order to process the exception.

When the NMI exception is recognized, all register contents are preserved with the following exceptions:

- *ErrorEPC* register, which contains the restart PC, and *Cause.BD2* which records whether the NMI was recognized in a branch delay slot.
- *Status.ERL* and *Status.BEV* flags are both set to 1.
- *Cause.EXC2* is set to 1 (NMI).

Servicing

Note that the NMI service routine entry address does not depend on the *Status.BEV* flag. In fact, the *Status.BEV* bit is unconditionally set to 1 before the NMI handler is entered. It is up to the NMI service routine to restore the setting of the *Status.BEV* bit prior to exit.

5.5.4 Performance Counter Exception

Cause

A lower-case performance counter exception occurs when a Performance counter overflows and conditions are met as described in Section 9.3.2. This exception is maskable by setting *Status.ERL* bit.

Exception Level: 2

Vector Address: 0x8000 0080 (DEV = 0), 0xBFC0 0280 (DEV = 1)

Processing

The value of *Cause.EXC2* is set to 2 (*PerfC)*. The *ErrorEPC* register contains the address of the instruction where the Performance counter exception was detected unless it is in a branch delay slot, in which case the *ErrorEPC* register contains the address of the preceding branch instruction and the *Cause.BD2* is set.

Servicing

When this exception is recognized, control is transferred to the applicable service routine.

5.5.5 Debug Exception

Cause

A DEBUG exception occurs whenever hardware breakpoint conditions as described in Chapter 13 are detected. This exception is maskable by setting *Status.ERL* bit.

Exception Level: 2

Vector Address: 0x8000 0100 (DEV = 0), 0xBFC0 0300 (DEV = 1)

Processing

The value of *Cause.EXC2* is set to 3 (*Dbg).* The *ErrorEPC* register contains the address of the instruction where the debug exception was detected unless it is in a branch delay slot, in which case the *ErrorEPC* register contains the address of the preceding branch instruction and *Cause.BD2* is set. Note that the Load data value breakpoint exception is imprecise. That is, the instruction where the breakpoint is detected is not the load instruction that triggers the breakpoint; see Chapter 13 for more details.

Servicing

When this exception is recognized, control is transferred to the applicable service routine.

5.5.6 Address Error Exception

Cause

The Address Error exception occurs when an attempt is made to execute one of the following:

- load or store a doubleword that is not aligned on a doubleword boundary
- load, fetch, or store a word that is not aligned on a word boundary
- load or store a halfword that is not aligned on a halfword boundary
- reference the kernel address space from User or Supervisor mode
- reference the supervisor address space from User mode

This exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of *Cause.ExcCode* is set to 4 (*AdEL*) or 5 (*AdES*), depending on whether the exception was caused due to an instruction reference (*AdEL*), load operation (*AdEL*), or store operation (*AdES*).

When this exception is recognized, the virtual address that was not properly aligned or that referenced protected address space is stored in the *BadVAddr* register. This update occurs even if the exception occurs within a level 1 or level 2 exception handler. The contents of the *VPN* field of the *Context* and *EntryHi* registers are undefined, as are the contents of the *EntryLo* register.

The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot. If it is in a branch delay slot, the *EPC* register contains the address of the preceding branch instruction and *Cause.BD* is set to indicate that the branch delay slot instruction actually caused the exception.

5.5.7 TLB Refill Exception

Cause

The TLB refill exception occurs when there is no TLB entry to match a reference to a mapped address space. This exception is not maskable.

Exception Level: 1

Vector Address: EXL = 0: 0x8000 0000 (BEV = 0), 0xBFC0 0200 (BEV = 1)
EXL = 1: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of *Cause.ExcCode* is set to either a value of 2 (TLBL) or 3 (TLBS). This code indicates whether the exception was caused due to an instruction reference, load operation, or store operation.

When this exception is recognized, the *BadVAddr*, *Context* and *EntryHi* registers are updated to hold the virtual address that failed address translation. The *EntryHi* register also contains the ASID for which the translation fault occurred. These actions take place even if the exception is recognized within a level 1 or level 2 exception handler. The *Random* register normally contains a valid location in which to place the replacement TLB entry. The contents of the *EntryLo* register are undefined. The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and *Cause.BD* is set.

The *EPC* register and *BD* bit in the *Cause* register point to the address of the instruction causing the exception.

Servicing

To service this exception, the contents of the *Context* register are used as a virtual address to fetch memory locations containing the physical page frame and access control bits for a pair of TLB entries. The two entries are placed into the *EntryLo0/EntryLo1* register; the *EntryHi* and *EntryLo* registers are then written into the TLB.

It is possible that the virtual address used to obtain the physical address and access control information is on a page that is not resident in the TLB. This condition is processed by allowing a TLB refill exception in the TLB refill handler. This second exception goes to the common exception vector because the *EXL* bit of the *Status* register is set.

5.5.8 TLB Invalid Exception

Cause

The TLB invalid exception occurs when a virtual address reference matches a TLB entry that is marked invalid (TLB valid bit cleared). This exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of *Cause.ExcCode* is set to either 2 (TLBL) or 3 (TLBS). This code indicates whether the exception was caused due to an instruction reference, load operation, or store operation.

When this exception is recognized, the *BadVAddr*, *Context,* and *EntryHi* registers are loaded with the virtual address that failed address translation. The *EntryHi* register also contains the ASID for which the translation fault occurred. These actions occur even if the exception is recognized within a level 1 or level 2 exception handler. The *Random* register normally contains a valid location in which to put the replacement TLB entry. The contents of the *EntryLo* register is undefined.

The *EPC* register contains the address of the instruction that caused the exception unless this instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

A TLB entry is typically marked invalid when one of the following is true:

- a virtual address does not exist
- the virtual address exists, but is not in main memory (a page fault)
- a trap is desired on any reference to the page (for example, to maintain a reference bit)

After servicing the cause of a TLB Invalid exception, the TLB entry is located with TLBP (TLB Probe), and replaced by an entry with that entry's *Valid* bit set.

5.5.9 TLB Modified Exception

Cause

The TLB modified exception occurs when a store operation generates a virtual address that matches a TLB entry that is marked valid but is not dirty and therefore is not writable. This exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of *Cause.ExcCode* is set to 1 (Mod) and the *BadVAddr, Context,* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi* register also contains the ASID for which the translation fault occurred. These actions occur even if the exception is recognized within a level 1 or level 2 exception handler. The contents of the *EntryLo* register is undefined.

The *EPC* register contains the address of the instruction that caused the exception unless that instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

The kernel uses the failed virtual address or virtual page number to identify the corresponding access control information. The page identified may or may not permit write accesses; if writes are not permitted, a write protection violation occurs.

If write accesses are permitted, the page frame is marked dirty/writable by the kernel in its own data structures. The *TLBP* instruction places the index of the TLB entry that must be altered into the *Index* register. The *EntryLo* register is loaded with a word containing the physical page frame and access control bits (with the *D* bit set), and the *EntryHi* and *EntryLo* registers are written into the TLB.

5.5.10 Bus Error Exception

Cause

A Bus Error exception is raised when *BUSERR** signal is asserted during bus transactions. This exception is masked when *Status.BEM, Status.EXL* or *Status.ERL* are set to 1.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of *Cause.ExcCode* is set to 6 (IBE) or 7 (DBE), indicating whether the exception was caused due to an instruction reference (*IBE*), load operation (*DBE*), or store operation (*DBE*). The *BadPAddr* is set to the physical address which caused a bus error when *Status.BEM* bit is 0.

The *EPC* register and *BD* bit in the *Cause* register point to the address of the instruction currently being executed by the processor.

Note that there is no necessary relationship between a bus error and the instruction being executed currently. For example, a bus error may be caused by instruction prefetch, or by a data cache line operation that is unrelated to any instruction. Furthermore, it could be caused by a load or store that was issued several instructions prior to the instruction that was executing when the bus error was recognized.

If a bus error is caused by a load or store instruction, the instruction is retired. If the instruction is a store, the nature of how memory is updated depends on the memory subsystem's design. If the instruction is a load, the value loaded into the destination register is indeterminate. If a data value breakpoint is pending for the memory address accessed, breakpoint recognition is implementation dependent.

Servicing

In the C790 the bus error exception is imprecise and as such difficult to recover from and continue processing. If a bus error occurs during instruction or data cache refills, the cache line loaded has undefined values in it. Since it is not possible in general to determine the offending address (from the *EPC*) the entire data and instruction cache contents should be invalidated by using Index Invalidate suboperation of the *CACHE* instruction. (See the *CACHE* instruction's definition for details on how to do this.)

5.5.11 System Call Exception

Cause

A SYSCALL exception occurs as a result of executing the *SYSCALL* instruction. This exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of *Cause.ExcCode* is set to 8 (Sys). The *EPC* register contains the address of the *SYSCALL* instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and *Cause.BD* is set.

Servicing

When this exception is recognized, control is transferred to the applicable system routine.

To resume execution, the *EPC* register must be altered so that the *SYSCALL* instruction does not re-execute; this is accomplished by adding a value of 4 to the *EPC* register (*EPC* register + 4) before returning.

If a *SYSCALL* instruction is in a branch delay slot, a more complicated algorithm, beyond the scope of this description, may be required.

5.5.12 BREAK Instruction Exception

Cause

A BREAK exception occurs as a result of executing the *BREAK* instruction. This exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of *Cause.ExcCode* is set to *9 (Bp).* The *EPC* register contains the address of the *BREAK* instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and *Cause.BD* is set.

Servicing

When a BREAK exception is recognized, control is transferred to the applicable system routine. Additional distinctions can be made by analyzing the unused bits of the *BREAK* instruction (bits 25:6), and loading the contents of the instruction whose address the *EPC* register contains. A value of 4 must be added to the contents of the *EPC* register (*EPC* register + 4) to locate the instruction if it resides in a branch delay slot.

To resume execution, the *EPC* register must be altered so that the *BREAK* instruction does not re-execute; this is accomplished by adding a value of 4 to the *EPC* register (*EPC* register + 4) before returning.

If a *BREAK* instruction is in a branch delay slot, interpretation of the branch instruction is required to resume execution.

5.5.13 Reserved Instruction Exception

Cause

The Reserved Instruction exception occurs when one of the following conditions occurs:

- an attempt is made to execute an instruction with an undefined major opcode (bits 31:26)
- an attempt is made to execute a SPECIAL instruction with an undefined minor opcode (bits 5:0)
- an attempt is made to execute a REGIMM instruction with an undefined minor opcode (bits 20:16)
- an attempt is made to execute a MMI instruction with an undefined minor opcode (bits 10:0)
- an attempt is made to execute a COPz instruction with an undefined minor opcode (bits 25:21)

Note: In the C790, 64-bit operations are always valid in User, Supervisor, and Kernel mode.

This exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of *Cause.ExcCode* is set to *10 (RI).* The *EPC* register contains the address of the reserved instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

5.5.14 Coprocessor Unusable Exception

Cause

The Coprocessor Unusable exception occurs when an attempt is made to execute a coprocessor instruction for either:

- a corresponding coprocessor unit that has not been marked usable via the *Status.Cu[ ]* bits or
- COP0 instructions, when the unit has been marked not usable and the process executes in either User or Supervisor mode.

NOTE: COP0 instructions always execute in Kernel mode, regardless of the setting of *Status.CU[0]*. Also note that the operation of the COP0 instructions EI and DI is not controlled by *Status.CU[0]*. Instead, the *Status.EDI* bit specifies whether the EI and DI instructions execute in User and Supervisor modes. In case execution is suppressed, EI and DI behave as no-operations in User and Supervisor modes; they do not signal an exception.

The exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of *Cause.ExcCode* is set to 11 *(CpU)* and the field *Cause.CE (Coprocessor Usage Error)* is set to indicate which of the four coprocessors was referenced. The *EPC* register contains the address of the unusable coprocessor instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

Servicing

The coprocessor unit to which an attempted reference was made is identified by the *CE* (Coprocessor Usage Error) field, which result in one of the following situations:

- If the process is entitled access to the coprocessor, the coprocessor is marked usable and the corresponding user state is restored to the coprocessor.
- If the process is entitled access to the coprocessor, but the coprocessor does not exist or has failed, interpretation of the coprocessor instruction is possible.
- If the *BD* bit is set in the *Cause* register, the branch instruction must be interpreted; then the coprocessor instruction can be emulated and execution resumed with the *EPC* register advanced past the coprocessor instruction.

5.5.15 Interrupt Exception

Cause

The Interrupt exception occurs when one of the three interrupt signals is asserted. The significance of the interrupts is dependent upon the specific system implementation.

Each of the three interrupts can be masked by clearing the corresponding bit in the *Int-Mask* field of the *Status* register, and all of the three interrupts can be masked at once by clearing the *IE* bit or EIE bit of the *Status* register.

All three interrupts are also masked at once when the *EXL* or *ERL* bit of the *Status* register is set to 1.

Interrupt IP[7] is set when the *Count* register is equal to the *Compare* register.

Exception Level: 1

Vector Address: 0x8000 0200 (BEV = 0), 0xBFC0 0400 (BEV = 1)

Processing

The value of *Cause.ExcCode* is set to 0 *(Int)*. The *IP* field of the *Cause* register indicates current interrupt requests. It is possible that more than one of the bits can be simultaneously set (or even *no* bits may be set) if the interrupt is asserted and then deasserted before this register is read.

Servicing

If the interrupt is hardware-generated, the interrupt condition is cleared by correcting the condition causing the interrupt pin to be asserted.

Due to the on-chip write buffer, a store to an external device (possibly clearing the interrupt) may not occur until after other instructions in the pipeline finish. Hence, the user must ensure that the store will occur before the *return from exception* instruction (*ERET*) is executed. This can be insured by executing a *SYNC* instruction. Otherwise the interrupt may be serviced again even though there is no actual interrupt pending.

5.5.16 SIO Exception

Cause

The SIO exception occurs when the ***SIOInt*** signal is asserted. This exception is maskable by setting *Status.ERL* bit.

Exception Level: 2

Vector Address: 0x8000 0100 (DEV = 0), 0xBFC0 0300 (DEV = 1)

Processing

The value of *Cause.EXC2* is set to 3(Dbg). The *Cause.SIOP* is set to 1. The *ErrorEPC* register contains the address of the instruction where the SIO exception was detected unless if is in a branch delay slot, in which case the *ErrorEPC* register contains the address of the preceding branch insruction and *Cause.BD2* is set.

Servicing

When this exception is recognized, control is transferred to the applicable service routine.

5.5.17 Integer Overflow Exception

Cause

An Integer Overflow exception occurs when an *ADD*, *ADDI*, *SUB*, *DADD*, *DADDI* or *DSUB* instruction results in a 2's complement overflow. This exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of *Cause.ExcCode* is set to 12 (Ov). The *EPC* register contains the address of the instruction that caused the exception unless the instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

5.5.18 Trap Exception

Cause

The TRAP exception occurs when a *TGE*, *TGEU*, *TLT*, *TLTU*, *TEQ*, *TNE*, *TGEI*, *TGEIU*, *TLTI*, *TLTIU*, *TEQI*, or *TNEI* instruction results in a TRUE condition. This exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The value of *Cause.ExcCode* is set to 13 *(Tr)*. The *EPC* register contains the address of the instruction causing the exception unless the instruction is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and *Cause.BD* is set.

5.5.19 Floating-Point Exception

Cause

The Floating-Point exception is used by the floating-point coprocessor. This exception is not maskable.

Exception Level: 1

Vector Address: 0x8000 0180 (BEV = 0), 0xBFC0 0380 (BEV = 1)

Processing

The common exception vector is used for this exception, and the FPE code in *Cause* register is set.

The contents of the Floating-Point Control/Status register indicate the cause of this exception.

This exception is cleared by clearing the appropriate bit in the Floating-Point Control/Status register.

For an unimplemented instruction exception, the kernel should emulate the instruction; for other exceptions, the kernel should pass the exception to the user program that caused the exception.

6. Memory Management

The C790 processor provides a memory management unit (MMU) which uses an on-chip translation look-aside buffer (TLB) to translate virtual addresses into physical addresses.

The C790 supports the MIPS compatible *32-bit* address and *64-bit* data mode. *Only* 32-bit virtual and physical addresses have been implemented. There is no requirement for address sign extension and address error exception checking will not be done on the “upper” 32-bits (which are ignored). The only condition that will generate the address error exception will be address alignment errors and segment protection errors. In Kernel mode, there will be address error exception free program counter wrap-around from *kseg3* to *kuseg*.

Since there is only one addressing mode, all the four MIPS ISAs (I, II, III, IV) and the C790 specific ISA are available without any restrictions in all of the three processor modes (with the appropriate MIPS ISA coprocessor usable restrictions). As such the reserved instruction (RI) exception will occur only when the processor really tries to execute an undefined opcode.

This chapter describes the processor virtual and physical address spaces, the virtual-to-physical address translation, the operation of the TLB in making these translations, and those System Control Coprocessor (COP0) registers that provide the software interface to the TLB.

6.1 Translation Look-aside Buffer (TLB)

Mapped virtual addresses are translated into physical addresses using an on-chip TLB. The TLB is a fully associative memory that holds 48 entries, which provide mapping to 48 odd / even page pairs (96 pages). When address mapping is indicated, each TLB entry is checked simultaneously for a match with the virtual address that is extended with an ASID stored in the low 8 bits of the *EntryHi* register.

The address mapped to a page ranges in size from 4 KB to 16 MB, in multiples of four; that is, 4K, 16K, 64K, 256K, 1M, 4M, 16M.

6.1.1 Translation Status

In C790 processor, as the one implemented in R4000, each TLB entry holds two sets of mapping information for two odd/even page pair and therefore the translation result is categorized into three states, hit, miss and invalid.

Upon address translation, if there is no virtual address match in all 48 entries, the translation result is categorized as TLB miss.
In this case, an exception is taken and software refills the TLB from the page table resident in memory. Software can write over a selected TLB entry or use a hardware mechanism to write into a random entry.

If there is a match on translation, the following takes place in the TLB hardware.

1. The translation information for odd page and even page is read out of the matching entry. Also the page size is extracted at the same time.
2. The TLB selects either of translation information in accordance with the page size information extracted above and the virtual address.
   This becomes the translation result in the TLB.

The translation result includes a valid flag to indicate the translation information is valid or not. If the flag is marked as ‘valid’, the translation is handled as TLB hit. The physical page number is extracted from the TLB and concatenated with the offset to form the physical address (see Figure 6-1).

If the flag is marked as ‘invalid’, the translation result is recognized as TLB invalid. In this case, an exception is taken to request the software to update the entry that got a match upon translation, by probing the TLB using *TLBP* operation.

6.1.2 Multiple Matches

Multiple match is the condition that there are two or more entries that match upon address translation. This is strictly prohibited and software is expected never to allow this to occur.
The C790 processor does NOT provide any meanings to detect this in hardware, such as TLB shutdown. The result of this condition is undefined and the further execution may provide incorrect result.

6.2 Address Spaces

This section describes the virtual and physical address spaces and the manner in which virtual addresses are converted or "translated" into physical addresses in the TLB.

6.2.1 Virtual Address Space

The C790 only implements 32 bits of virtual address space. There is no requirement for address sign extension and no checking will be done on the upper 32 bits of the address.

Figure 6-1 shows the translation of a virtual address into a physical address.

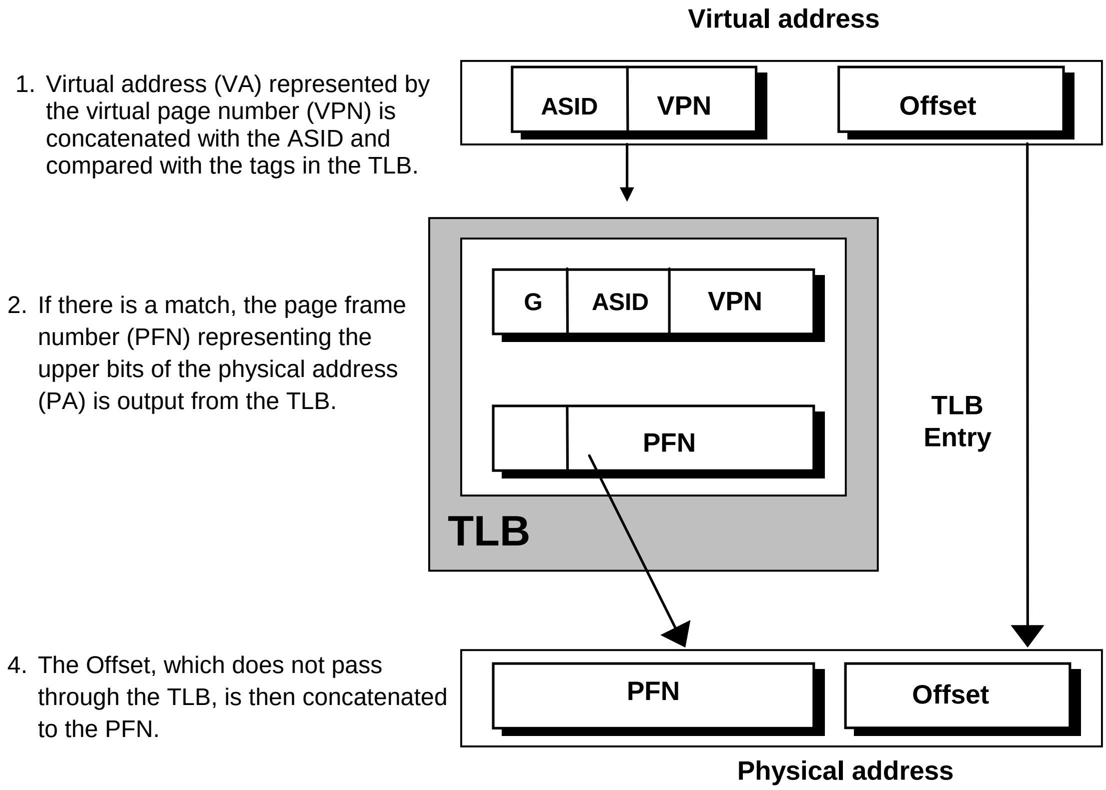


Figure 6-1. Overview of a Virtual-to-Physical Address Translation

As shown in Figure 6-2, the virtual address is extended with an 8-bit address space identifier (ASID), which reduces the frequency of TLB flushing when switching contexts. This 8-bit ASID is in the COP0 *EntryHi* register as described later in this chapter.

6.2.2 Physical Address Space

Using a 32-bit address, the processor physical address space encompasses 4 GB. The following section describes the translation of a virtual address to a physical address.

6.2.3 Virtual-to-Physical Address Translation

Converting a virtual address to a physical address begins by comparing the virtual address from the processor with the virtual addresses in the TLB; there is a match when the virtual page number (VPN) of the address is the same as the VPN field of the entry, and either:

- the Global (G) bit of the TLB entry is set, or
- the ASID field of the virtual address (taken from the 8-bit ASID field of the EntryHi register) is the same as the ASID field of the TLB entry.

If there is no match, a TLB Miss exception is taken by the processor and software can refill the TLB from a page table of virtual / physical addresses in memory.

If there is a virtual address match in the TLB, the physical address is output from the TLB and concatenated with the *Offset*, which represents an address within the page frame space. The *Offset* does not pass through the TLB. At the same time, the valid bit output from TLB is checked to qualify the translation. If this bit is not set, a TLB Invalid exception is taken by the processor and software can update the TLB.

Virtual-to-physical translation is described in greater detail throughout the remainder of this chapter. Figure 6-9, shown at the end of this chapter, is a detailed flow diagram of this process.

6.2.4 32-bit Address Translation Mode

The C790 supports only 32-bit address translation mode. 64-bit addressing mode is *not* supported.

Figure 6-2 shows the virtual-to-physical address translation of a 32-bit address.

- The top portion of Figure 6-2 shows a virtual address with a 12-bit, or 4-KB, page size, labeled *Offset*. The remaining 20 bits of the address represent the VPN, and index the 1M-entry page table.
- The bottom portion of Figure 6-2 shows a virtual address with a 24-bit, or 16-MB, page size, labeled *Offset*. The remaining 8 bits of the address represent the VPN, and index the 256-entry page table.

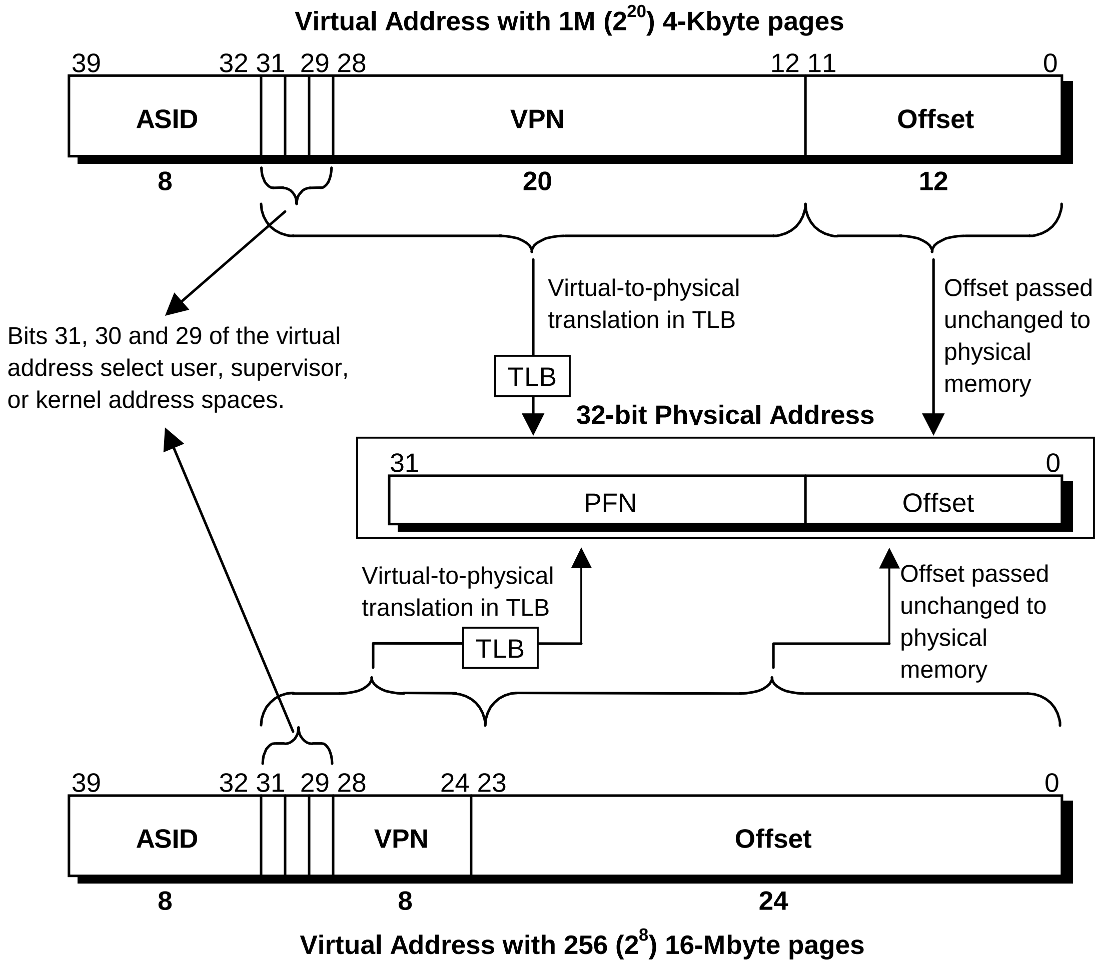


Figure 6-2. 32-bit Mode Virtual Address Translation

6.2.5 Operating Modes

The processor has the three standard MIPS operating modes:

- User mode
- Supervisor mode
- Kernel mode

Selection between the three modes can be made by the operating system (when in Kernel mode) by writing into *Status* register's KSU field. The processor is forced into Kernel mode when the processor is handling a Level 1 exception (the EXL bit is set - also called the Exception Level mode in R-series processors) or a Level 2 exception (the ERL bit is set - also called the Error Level mode in R-series processors).

In the following table, dashes represent 'don't cares'.

Table 6-1 Processor Modes

Description	KSU	ERL	EXL
32-bit User mode	10	0	0
32-bit Supervisor mode	01	0	0
32-bit Kernel mode	00	0	0
32-bit Kernel mode (Level 1 exception)	-	0	1
32-bit Kernel mode (Level 2 exception)	-	1	-

Figure 6-3 shows a state transition among these three modes.

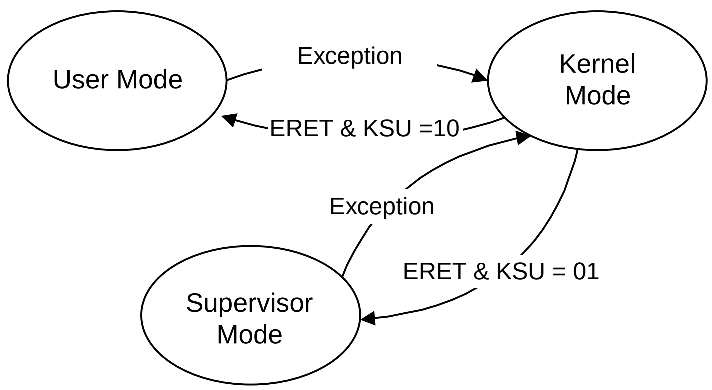


Figure 6-3 State Transition among Operating Modes

Table 6-2 summarizes address space for each operating mode.

Table 6-2. Address Space

Virtual Address	32-bit User Mode	32-bit Supervisor Mode	32-bit Kernel Mode
0xFFFF FFFF to 0xE000 0000	Address Error	Address Error	*kseg3* (0.5 GB) Mapped
0xDFFF FFFF to 0xC000 0000		*sseg* (0.5 GB) Mapped	*ksseg* (0.5 GB) Mapped
0xBFFF FFFF to 0xA000 0000		Address Error	*kseg1* (0.5 GB) Unmapped* Uncached
0x9FFF FFFF to 0x8000 0000			*kseg0* (0.5 GB) Unmapped* Cached**
0x7FFF FFFF to 0x0000 0000	*useg* (2 GB) Mapped	*suseg* (2 GB) Mapped	*kuseg* (2 GB) Mapped (becomes unmapped if ERL is 1)

*Note: Virtual addresses of Kernel segments, *kseg0* and *kseg1*, are not mapped through the TLB and always translated into physical addresses from 0x0000 0000 to 0x1FFF FFFF.

** Note: The kseg0 cache algorithm is controlled by the K0 field in the Config register.

6.2.6 User Mode Operations

In User mode, a single, uniform virtual address space, labeled User segment, is available; its size is:

- 2 GB (2^{31} bytes) (*useg*)

Figure 6-4 shows User mode virtual address space.

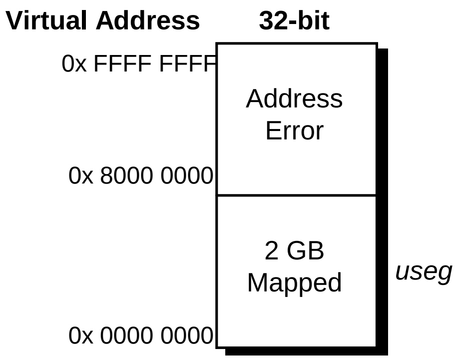


Figure 6-4. User Mode Virtual Address Space

The User segment starts at address 0x0000 0000 and the current active user process resides in *useg*. The TLB identically maps all references to *useg* from all modes, and controls cache accessibility.

The processor operates in User mode when the *Status* register contains the following bit-values:

- *KSU* bits = 10_2
- *and EXL* = 0
- *and ERL* = 0

Table 6-3 lists the characteristics of the User mode segment, *useg* .

Table 6-3. User Mode Segments

Address Bit Values	Status Register Bit Values			Segment Name	Virtual Address Range	Segment Size
	KSU	EXL	ERL			
A[31] = 0	10_2	0	0	*useg*	0x0000 0000 through 0x7FFF FFFF	2 Gbyte (2^{31} bytes)

User Mode, User Space(*useg*)

In User mode(*KSU* = 10_2 in the *Status* register), when the most-significant bit of the 32-bit virtual address is set to 0, the *useg* virtual address space is selected; it covers the 2^{31} bytes (2 GB) of the current user address space. All valid User mode virtual addresses have their most-significant bit cleared to 0; any attempt to reference an address with the most-significant bit set while in User mode causes an Address Error exception.

The system maps all references to *useg* through the TLB. Bit settings within the TLB entry for the page determine the cacheability of a reference. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space starts at virtual address 0x0000 0000 and runs through 0x7FFF FFFF.

6.2.7 Supervisor Mode Operations

Supervisor mode is designed for layered operating systems in which a true kernel runs in C790 Kernel mode, and the rest of the operating system runs in Supervisor mode.

The processor operates in Supervisor mode when the *Status* register contains the following bit-values:

- *KSU* = 01_2
- *and EXL* = 0
- *and ERL* = 0

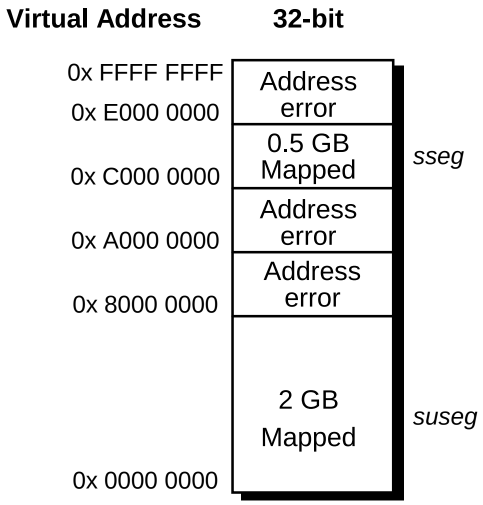


Figure 6-5. Supervisor Mode Virtual Address Space

Table 6-4. Supervisor Mode Segments

Address Bit Values	Status Register Bit Values			Segment Name	Virtual Address Range	Segment Size
	KSU	EXL	ERL			
A[31] = 0	01_2	0	0	*suseg*	0x0000 0000 through 0x7FFF FFFF	2 Gbyte (2^{31} bytes)
A[31:29] = 110_2	01_2	0	0	*sseg*	0xC000 0000 through 0xDFFF FFFF	0.5 Gbyte (2^{29} bytes)

Supervisor Mode, User Space (*suseg*)

In Supervisor mode (*KSU* = 01_2 in the *Status* register), when the most-significant bit of the 32-bit virtual address is set to 0, the *suseg* virtual address space is selected; it covers the 2^{31} bytes (2 Gbytes) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space starts at virtual address 0x0000 0000 and runs through 0x7FFF FFFF.

Supervisor Mode, Supervisor Space (*sseg*)

In Supervisor mode (*KSU* = 01_2 in the *Status* register), when the three most-significant bits of the 32-bit virtual address are 110_2, the *sseg* virtual address space is selected; it covers 2^{29}-bytes (512 Mbytes) of the current supervisor address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space begins at virtual address 0xC000 0000 and runs through 0xDFFF FFFF.

6.2.8 Kernel Mode Operations

The processor operates in Kernel mode when the *Status* register contains one of the following values:

- *KSU* = 00_2
- *or EXL* = 1
- *or ERL* = 1

The processor enters Kernel mode whenever an exception is detected and it remains in Kernel mode until an Exception Return (*ERET*) instruction is executed. The *ERET* instruction restores the processor to the mode existing prior to the exception.

Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 6-6.

Table 6-5 lists the characteristics of the kernel mode segments.

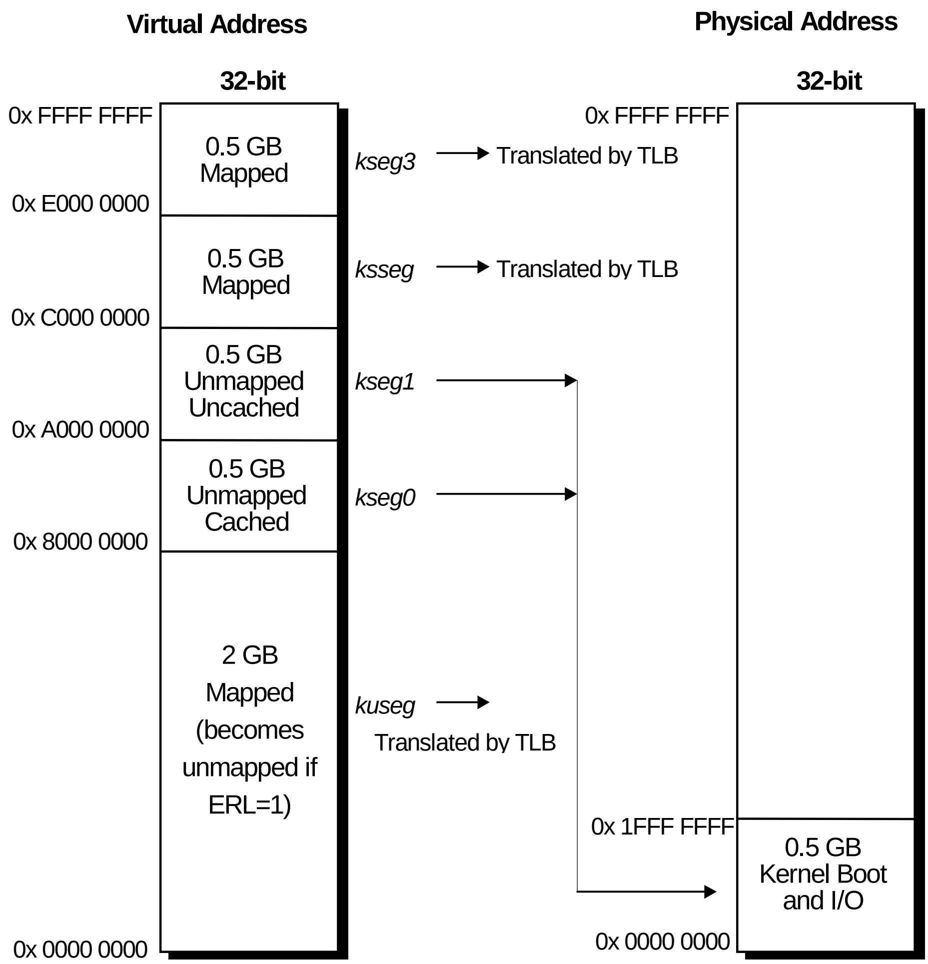


Figure 6-6. Kernel Mode Address Space

Table 6-5. Kernel Mode Segments

Address Bit Values	Status Register Bit Values KSU | EXL | ERL	Segment Name	Virtual Address Range	Segment Size
A[31] = 0	KSU = 00_2	*kuseg*	0x0000 0000 through 0x7FFF FFFF	2 Gbyte (2^{31} bytes)
A[31:29] = 100_2	*or*	*kseg0*	0x8000 0000 through 0x9FFF FFFF	0.5 Gbyte (2^{29} bytes)
A[31:29] = 101_2	EXL = 1	*kseg1*	0xA000 0000 through 0xBFFF FFFF	0.5 Gbyte (2^{29} bytes)
A[31:29] = 110_2	*or*	*ksseg*	0xC000 0000 through 0xDFFF FFFF	0.5 Gbyte (2^{29} bytes)
A[31:29] = 111_2	ERL = 1	*kseg3*	0xE000 0000 through 0xFFFF FFFF	0.5 Gbyte (2^{29} bytes)

Kernel Mode, User Space (*kuseg*)

In Kernel mode (*KSU* = 00_2 or EXL = 1 or ERL = 1 in the *Status* register), when the most-significant bit of the virtual address, A[31], is a 0, the 32-bit *kuseg* virtual address space is selected; it covers the full 2^{31} bytes (2 GB) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

When ERL = 1 in the *Status* register, the user address, *kuseg*, region becomes a 2^{31}-byte unmapped, uncached address space (that is, mapped directly to physical addresses 0x0000 0000 through 0x7FFF FFFF).

Kernel Mode, Kernel Space 0 (*kseg0*)

In Kernel mode (*KSU* = 00_2 or EXL = 1 or ERL = 1 in the *Status* register), when the most-significant three bits of the virtual address are 100_2, 32-bit *kseg0* virtual address space is selected; it is the 2^{29}-byte (512 MB) kernel physical space.

References to *kseg0* are not mapped through the TLB; the physical address selected is defined by subtracting 0x8000 0000 from the virtual address. The *K0* field of the *Config* register, described in this chapter, controls cacheability and coherency.

Kernel Mode, Kernel Space 1 (*kseg1*)

In Kernel mode (*KSU* = 00_2 or EXL = 1 or ERL = 1 in the *Status* register), when the most-significant three bits of the 32-bit virtual address are 101_2, 32-bit *kseg1* virtual address space is selected; it is the 2^{29}-byte (512 MB) kernel physical space.

References to *kseg1* are not mapped through the TLB; the physical address selected is defined by subtracting 0xA000 0000 from the virtual address.

Caches are disabled for accesses to these addresses, and physical memory (or memory-mapped I/O device registers) is accessed directly.

Kernel Mode, Supervisor Space (*ksseg*)

In Kernel mode (*KSU* = 00_2 in the *Status* register), when the most-significant three bits of the 32-bit virtual address are 110_2, the *ksseg* virtual address space is selected; it is the current 2^{29}-byte (512 MB) supervisor virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

Kernel Mode, Kernel Space 3 (*kseg3*)

In Kernel mode (*KSU* = 00_2 in the *Status* register), when the most-significant three bits of the 32-bit virtual address are 111_2, the *kseg3* virtual address space is selected; it is the current 2^{29}-byte (512 MB) kernel virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

6.3 System Control Coprocessor

The System Control Coprocessor (COP0) is implemented as an integral part of the CPU, and supports memory management, address translation, exception handling, and other privileged operations. The COP0 registers shown in Figure 6-7 plus a 48-entry TLB make up the MMU.

Each COP0 register has a unique number that identifies it; this number is referred to as the *register number*. For instance, the *PageMask* register is register number 5.

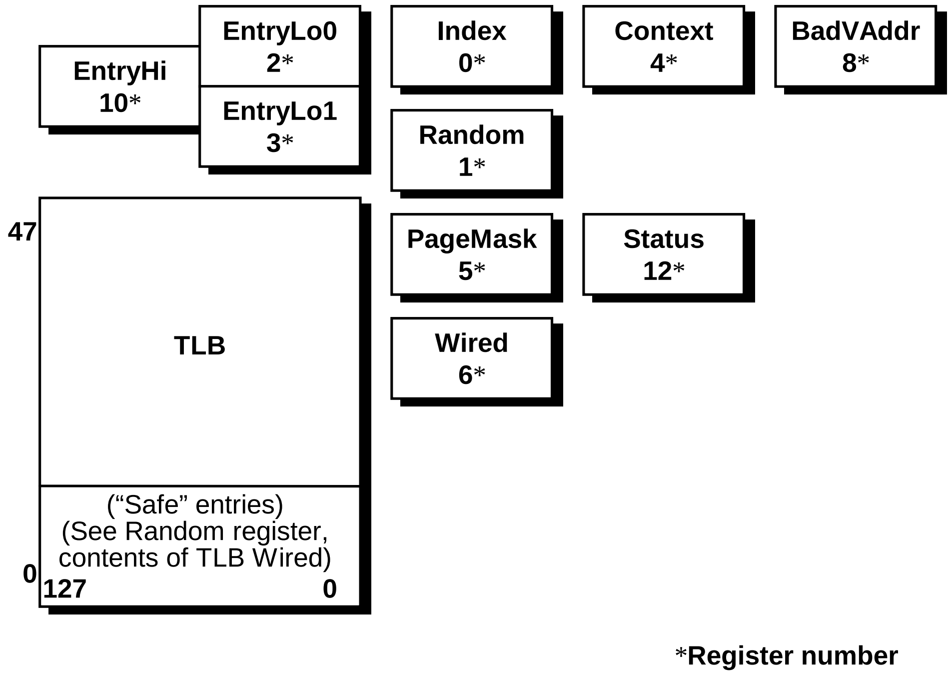


Figure 6-7. COP0 Registers and the TLB

6.3.1 Format of a TLB Entry

Figure 6-8 shows the TLB entry formats for the 32-bit address translation modes. Each field of an entry has a corresponding field in the *EntryHi*, *EntryLo0*, *EntryLo1*, or *PageMask* registers. For example, the *Mask* field of the TLB entry is also held in the *PageMask* register.

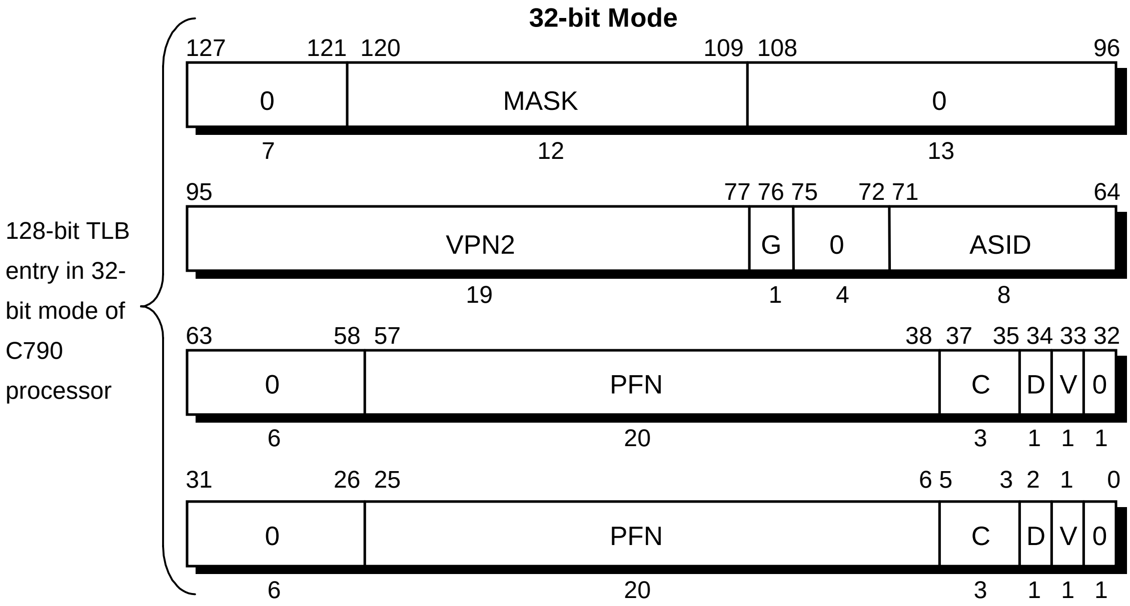


Figure 6-8. Format of a TLB Entry

The format of the *EntryHi*, *EntryLo, EntryLo1*, and *PageMask* registers are nearly the same as the TLB entry. The one exception is the *Global* field (*G* bit), which is used in the TLB, but is reserved in the *EntryHi* register. The following register tables describe the TLB entry fields shown in Figure 6-8.

PageMask Register

31–25	24–13	12–0
0	MASK	0
7	12	13

MASK	Page comparison mask.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

EntryHI Register

31–13	12–8	7–0
VPN2	0	ASID
19	5	8

VPN2	Virtual page number divided by two (maps to two pages).
ASID	Address space ID field. An 8-bit field that lets multiple processes share the TLB; each process has a distinct mapping of otherwise identical virtual page numbers.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

EntryLo0 Register

31–26	25–6	5–3	2	1	0
0	PFN	C	D	V	G
6	20	3	1	1	1

EntryLo1 Register

31–26	25–6	5–3	2	1	0
0	PFN	C	D	V	G
6	20	3	1	1	1

PFN	Page frame number; the upper bits of the physical address.
C	Specifies the TLB page coherency attribute; see Table 6-7.
D	Dirty. If this bit is set, the page is marked as dirty and, therefore, writable. This bit is actually a write-protect bit that software can use to prevent alteration of data.
V	Valid. If this bit is set, it indicates that the TLB entry is valid; otherwise, a TLB invalid exception occurs.
G	Global. If this bit is set in both LO0 and LO1, then the processor ignores the ASID during TLB lookup.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

The TLB page coherency attribute (*C*) bits specify whether references to the page should be either of cached, uncached, or uncache-accelerated. Table 6-6 shows the coherency attributes selected by the *C* bits.

Table 6-6 TLB Page Coherency (C) Bit Values

C[5:3] Value	Page Coherency Attribute
0	Reserved
1	Reserved
2	Uncached
3	Cacheable, write-back, write-allocate
4	Reserved
5	Reserved
6	Reserved
7	Uncached, Accelerated

Write-back with allocate fetches the line with the missed data both on load misses and on store misses. Therefore, storing data to such pages is always performed to the data cache and will not be sent to the write buffer.

Uncached accelerated data provides a special kind of acceleration for handling uncached data. On a load of an uncached accelerated data item (which can range in size from a byte to a quadword) the C790 will always fetch an aligned 128-byte quantity from memory. These eight quadwords will be placed in a special 128-byte buffer called the uncache accelerated buffer, or UCAB in the CPU. Any subsequent loads which "hit" the UCAB will get the data from the UCAB. This process reduces bus traffic. The UCAB will be invalidated under the following conditions:

- Any load operation which doesn't hit the buffer, or
- any store operation, or
- a SYNC (or SYNC.L) operation, or
- any exception.

For uncached accelerated stores, the C790 write-back buffer (128-bit x 8) also has some special features. On the first store of an uncached accelerated write the write-back buffer will mark the fact that this is an uncached accelerated write to a particular address. Subsequent uncached accelerated stores which hit within the same 128-bit address boundary will be accumulated (gathered) within the same write buffer entry. This process of data gathering reduces bus traffic. The gathering process will be terminated under the following conditions:

- Any store which can't be gathered (different attribute or different address), or
- any load operation, or
- a SYNC (or SYNC.L) operation, or
- any exception.

6.4 Virtual-to-Physical Address Translation Process

In the supported 32-bit mode, the highest 8 to 20 bits of the virtual address (depending upon the page size) are compared to the contents of the TLB virtual page number. The 8-bit ASID is only compared if the global bit, G, is not set.

If a TLB entry matches, the physical address and access control bits (*C, D*, and *V*) are retrieved from the matching TLB entry. While the *V* bit of the entry must be set for a valid translation to take place, it is not involved in the determination of a matching TLB entry.

Figure 6-9 illustrates the TLB address translation process.

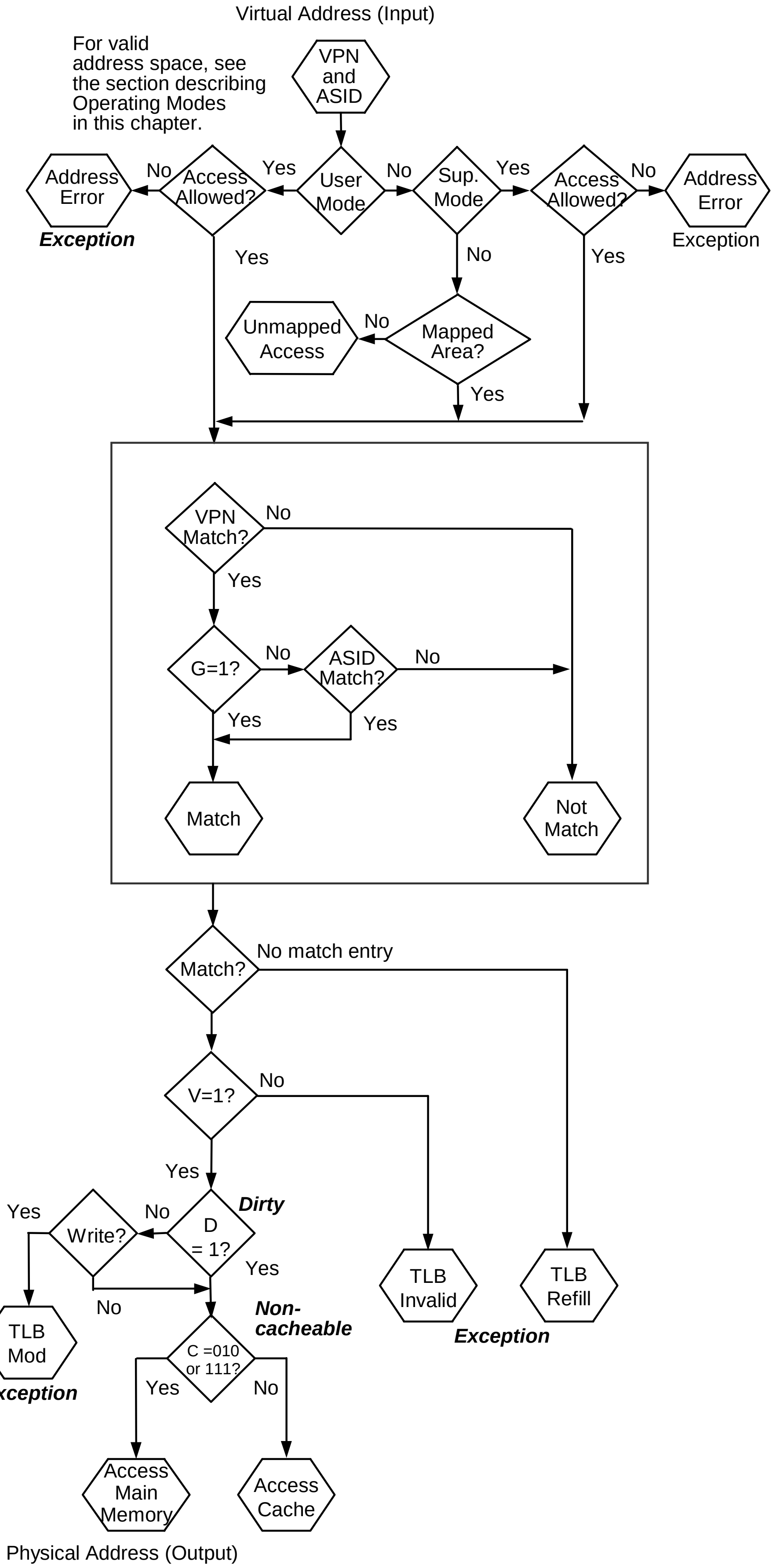


Figure 6-9. TLB Address Translation

If there is no TLB entry that matches the virtual address, a TLB miss exception occurs. If the access control bits (*D* and *V*) indicate that the access is not valid, a TLB modified or TLB invalid exception occurs.

If the *C* bits equal 010_2 (Uncached) or 111_2 (Uncached Accelerated), the physical address that is generated directly accesses main memory, bypassing the cache.

6.5 TLB Instructions

Table 6-7 lists the instructions that the CPU provides for working with the TLB. See Appendix C for a detailed description on these instructions.

Table 6-7. TLB Instructions

OpCode	Description of Instruction
TLBP	Translation Look-aside Buffer Probe
TLBR	Translation Look-aside Buffer Read
TLBWI	Translation Look-aside Buffer Write Index
TLBWR	Translation Look-aside Buffer Write Random

7. Caches

The C790 core contains both an instruction cache and a separate data cache. The processor also contains a small size of read only cache memory for uncached accelerated area.

This chapter describes the cache structures, operation of the caches, and cache control.

7.1 Cache Features

The two caches are configured as shown in Table 7-1:

Table 7-1. Cache Configuration

Cache	Size	Organization	Line Size	Refill Size
Instruction Cache	32 KB	2-Way	64 bytes	64 bytes
Data Cache	32 KB	2-Way	64 bytes	64 bytes

The following are the main features of the caches:

- Separate Instruction Cache and Data Cache
- Virtually indexed and physically tagged caches
- 64 Byte line size
- 64 Byte Refill size
- 2-way set-associative cache for higher performance
- Write-back policy for the Data Cache
- Missed quadword first sequential order burst refills for the Data Cache
- Data Cache line locking
- Non-Blocking Loads
- Data cache supports multiple Hits under a single miss
- No Snoop capability

No cache snoop capability has been provided. The user may choose to use *CACHE* instructions to keep coherency between caches and main memory.

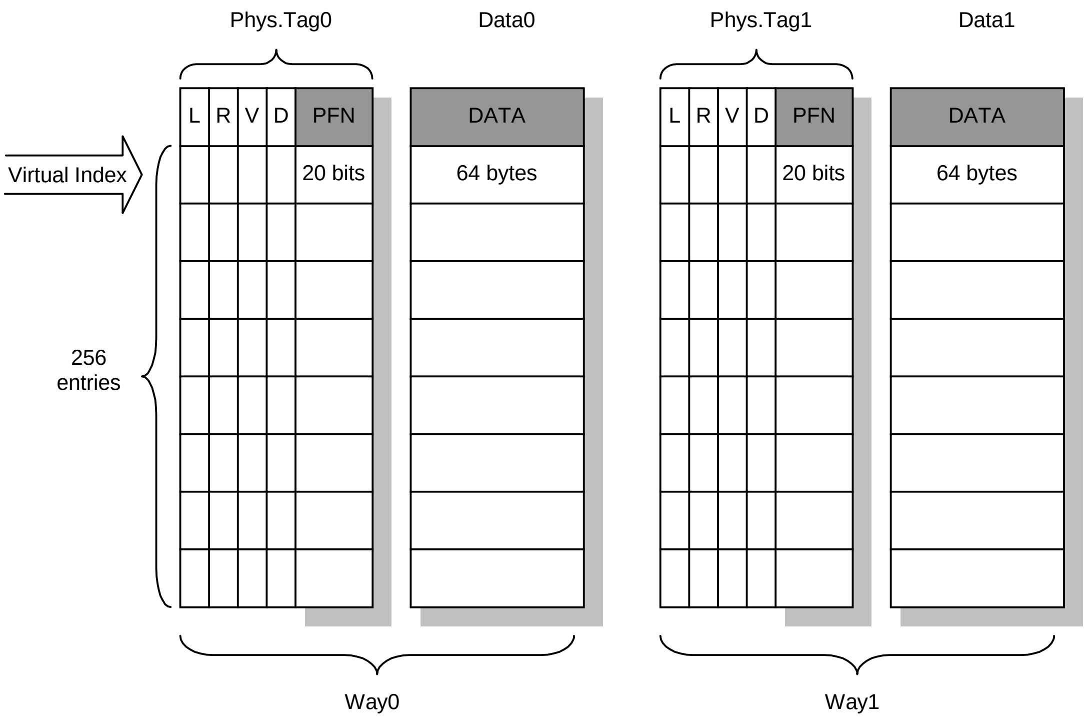
7.2 Organization of the Caches

Organization of the caches is illustrated in Figure 7-1 and Figure 7-2. Both the Instruction Cache and the Data Cacher are 2-way set-associative. Each cache line consists of a ***tag*** and ***data***. Each cache has a data line size of 64 bytes.

7.2.1 Data Cache

The Data Cache is connected to the CPU via a 128-bit bus. Therefore, the Data Cache can supply to the CPU or the coprocessors up to a quadword of data per access.

The following diagram shows Data Cache structure. Tags are discussed in detail in a later section.



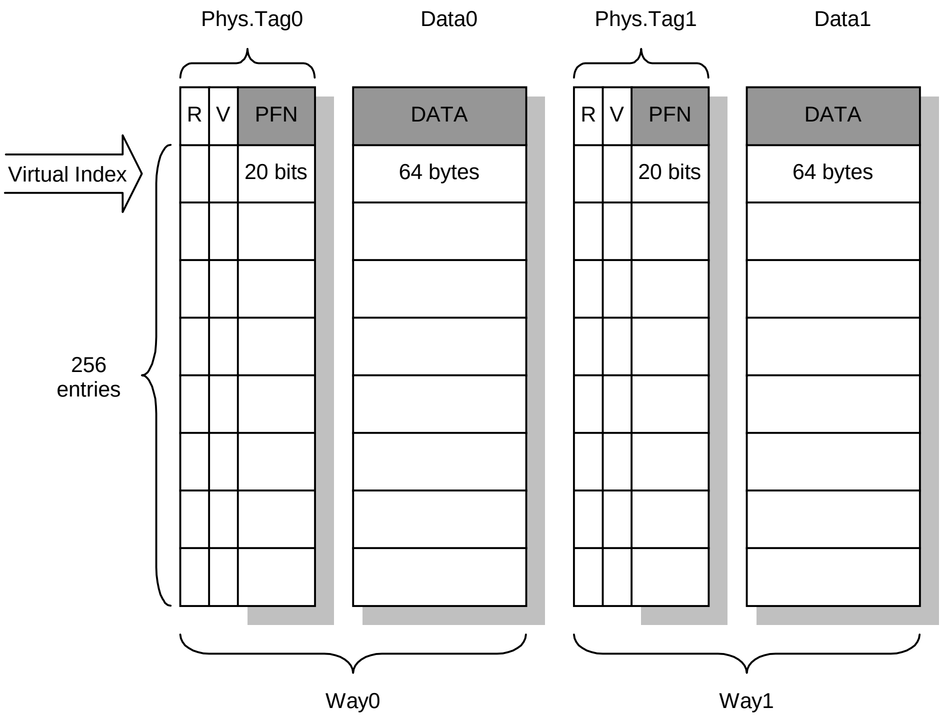
L	**Lock Bit**	**For description, see Section 7.3.7,** ***Data Cache Lock*** Function
R	**LRF Bit**	**For description, see Section 7.3.1,** ***Line Replacement Algorithm***
V	**Valid Bit**	**For description, see Section 7.2.3,** ***Tag Structure***
D	**Dirty Bit**	**For description, see Section 7.2.3,** ***Tag Structure***

Figure 7-1. Organization of Data Cache

7.2.2 Instruction Cache

The Instruction Cache is connected to the CPU pipeline via a 64-bit bus. This enables the CPU to fetch two instructions per cycle from the Instruction Cache.

The following diagram shows Instruction Cache structure. Tags are discussed in detail in a later section.



R **LRF Bit**

V **Valid Bit**

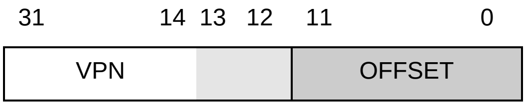
Figure 7-2. Organization of Instruction Cache

7.2.3 Tag Structure

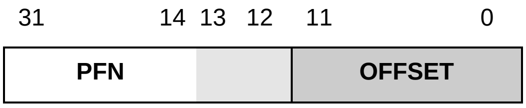
The general structure of a tag consists of a set of state bits and a physical page frame number or ***PFN*** field. The Data Cache and the Instruction Cache have different numbers of state bits; for more information, refer to the discussions in the following sections.

The size of the tag and the number of virtual address bits indexing the caches are dependent upon the size of the cache, address space, and set associativity. The C790 supports 32-bit virtual and physical addresses as shown in the figure below:

Virtual Address (VA)



Physical Address (PA)



Since the cache line size is fixed at 64 bytes, that is, four quadwords per entry, the Tag Cache associated with each way will have one tag for every four quadwords. Table 7-2 shows cache sizes, address bits and tag size.

Table 7-2. Cache Size and Access Bits

Cache	Size	Way	Size of Each Way	Cache Virtual Address Index Bits	Tag Cache Size of Each Way	Tag Virtual Address Index
Data	32 K	2 WAY	256 x 64 Bytes	13:4	256 x 20 Bits	13:6
Instruction	32 K	2 WAY	256 x 64 Bytes	13:3	256 x 20 Bits	13:6

While the caches are indexed by the virtual address, the tag comparison is physical. This is possible because the caches and the TLB are accessed in parallel. So, when the tags have been accessed, the page frame number is ready to be compared against the translated virtual address for a cache hit or miss.

C790 Programming Note:

Overlapping of the cache index bit range and PFN bit range causes the “cache aliasing problem”. C790 does not have any hardware mechanisms to detect the cache aliasing. It is programmer’s responsibility to avoid the cache aliasing. When a physical page is mapped on the different virtual pages, VPN[13:12] have to be same in both virtual address. The conservative way to avoid this is that VPN[13:12] == PFN[13:12] whenever a page is mapped.

7.2.3.1 Data Cache Tag Structure

In addition to the physical page frame number (PFN), each Data Cache Tag entry also contains additional ***Cache State*** bits as shown below. All lines in both ways of the Data Cache have these four state bits. Cache line state bits are also illustrated in Figure 7-1.

Data Cache Tag Fields

Dirty (D)	Valid (V)	LRF (R)	Lock (L)	PFN

Two state bits, *DIRTY* and *VALID*, together identify which of three states the Data Cache is in: Valid Clean, Valid Dirty, or Invalid. Table 7-3 shows the state of the Data Cache line as a function of *DIRTY* and *VALID* bits.

Table 7-3. Data Cache Line States

Dirty Bit (D)	Valid Bit (V)	Cache Line State
X	0	Invalid
0	1	Valid Clean
1	1	Valid Dirty

Even if Cache Instruction try to set V = 0, D = 1 state, Dirty bit is forced to zero in C790 implementation.

The *LRF* bit is the Least-Recently-Filled line replacement bit.

The *LRF* bits serve as a replacement algorithm between the two ways of the Data Cache. A refill access to a cache line in a way will flip the *LRF* bit to point to the other way as the least recently filled. For details of the LRF line update operation refer to Section 7.3.1.

As Figure 7-1 illustrates, Data Cache lines in each way have a *LOCK* bit. The *LOCK* bit, as explained in Section 7.3.7, *Data Cache Lock* Function, locks lines in one of the ways to keep data from being replaced.

7.2.3.2 Instruction Cache Tag Structure

In addition to the physical page frame number (PFN), each Instruction Cache Tag entry also contains two additional *Cache State* bits as shown below. All lines in both ways of the Instruction Cache have these two state bits.

Instruction Cache Tag Fields

Valid (V)	LRF (R)	PFN

The Instruction Cache *VALID* state bit defines whether each line is in the Valid or Invalid states.

The *LRF* bit is the Least-Recently-Filled line replacement bit. *LRF* bits serve as a replacement algorithm between the two ways of the Instruction Cache. A refill access to a cache line in a way will flip the *LRF* bit to point to the other way as the least recently filled. For details of LRF line update operation refer to Section 7.3.1.

7.2.4 State of Cache Tags After Reset

For all Data Cache tags the following fields are initialized to 0 upon reset:

- Valid
- Dirty
- LRF
- Lock

For all Instruction Cache tags the following fields are initialized to 0 upon reset:

- Valid
- LRF

All other fields in the Instruction Cache and the Data Cache contents are undefined upon reset.

7.3 Cache Operations

This section describes cache operation in regard to read/write policies, coherency, write-back policy, and the lock function.

7.3.1 Line Replacement Algorithm

The line replacement policy for both the Instruction Cache and the Data Cache is based on the Least Recently Filled (LRF) algorithm. In this policy, the LRF bit of a way is modified (inverted) only when a cache line refill occurs to the corresponding way. Load/store accesses to the Data Cache *do not* modify the LRF bit. The bit indicating which way is the least recently filled way is the XOR of the two LRF bits of the two ways of the cache.

Table 7-4. LRF Line Replacement Algorithm

Current Way0 LRF	Current Way1 LRF	XOR	Refill Way	New Way0 LRF	New Way1 LRF
0	0	0	0	1	0
1	0	1	1	1	1
1	1	0	0	0	1
0	1	1	1	0	0

The column under XOR indicates the way which could be refilled (line replaced) on the next refill at that line location. *Note that the table shown above is valid only when none of the ways of the cache line is locked. If a way of the cache line is locked, then regardless of the state of the LRF bits, the least recently filled way will always be the unlocked way.*

The behavior is also slightly different for Instruction and Data Caches when one of the way is invalid. For the Data Cache the algorithm is followed exactly as given above irrespective of the ways being valid or invalid. For the Instruction Cache the algorithm given above is followed as long as both the ways are valid. Once a way becomes invalid, then that way gets priority of being filled over the valid way irrespective of the LRF bits.

7.3.2 Non-blocking Loads and Hit Under Miss

The Data Cache supports **non-blocking load** and **hit under miss** to improve performance. When a Data Cache miss occurs or an uncached load instruction is issued, *Non-blocking load* allows the pipeline to continue instruction execution until one of the following occurs:

1. A subsequent non-load/store/pref instruction has data dependency with the load that is pending (to be retired).
2. A pipeline0 stalls.

Hit under miss is a feature that allows access (load or store) to the Data Cache while a previous load miss (cached, uncached or uncached accelerated), a previous store miss (cached) or a previous prefetch miss (cached) is still pending. In this case, access to the cache proceeds and the pipe does not stall.

Uncached loads also do not stall the pipeline while they are pending (to be retired). The pipeline continues instruction execution until one of the following occurs:

1. A subsequent load/store/pref instruction has data dependency with the load that is pending (to be retired).
2. A Data Cache miss occurs or a miss occurs on the Uncached Accelerated Buffer.
3. An Uncached load instruction is issued.

To summarize, *Non-blocking load* and *Hit under miss* allow the pipelene to continue instruction execution until one of following occurs when a Data Cache miss occurs or an uncached load instruction is issued:

1. A subsequent instruction has data dependency with the load that is pending (to be retired).
2. A Data Cache miss occurs or a miss occurs on the Uncached Accelerated Buffer.
3. An uncached load instruction is issued.
4. A pipeline0 stalls.

Loads to the *GPR*s (IU) and *FPR*s (FPU) all follow the non-blocking protocol (when it is enabled). Loads to COP1 is **always** blocking.

7.3.3 Cache Miss and Hit Operations

In case of a Data Cache hit, the cache provides data to the CPU in 128-bit (single quadword) quantities. In case of an Instruction Cache hit, the cache provides data ("instruction") in 64-bit quantities. CPU reads or writes to the Data Cache in quantities less than 128 bits are specified by the least significant four bits of the address, bits 3:0.

Cache misses are processed by the cache controller in 64-byte quantities - one cache line. Since the caches are connected to the system bus via a 128-bit bus, cache refill takes a burst of 4 bus cycles (8 CPU cycles) that is, four quadwords are transferred in 4 bus cycles (actual transfer time can be more due to bus arbitration etc). These reads are performed in sequential order for both the Instruction Cache and the Data Cache. The quadword for which the address missed is always fetched first.

Table 7-5 indicates the sequential order. PA[5:4] are two least-significant address bits that are put out on the CPU Bus. Figure 7-3 illustrates the case where the second quadword, shaded area, missed and shows the order in which data are read from main memory.

Table 7-5. Quadword Retrieved Address PA[5:4]

Bus Cycle	Starting Block Address PA[5:4]			
	00	01	10	11
1	00	01	10	11
2	01	10	11	00
3	10	11	00	01
4	11	00	01	10

	128 bits	128 bits	128 bits	128 bits
	11	10	01	00
Read order	Third	Second	First	Fourth

Figure 7-3. Read Missed Processed in Sequential Order

In case of a write miss to the Data Cache (for an allocate-on-write address), the cache controller will read in sequential order a cache line from main memory. Whether the cache line, being replaced, is first written out to memory or not - due to the *DIRTY* bit being set - is discussed in the next section.

The Instruction Cache processes cache misses in burst of 4 quadwords, just like the Data Cache. Furthermore, in case of an Instruction Cache miss, the pipeline starts in the same cycle the final quadword is stored into the Instruction Cache.

7.3.4 Data Cache Writeback Policy

Data cache lines are written back to the memory in the following cases:

1. The processor executes Index Write Back Invalidate CACHE instruction suboperation as defined in Appendix C and the line data are dirty. Or Hit Writeback Invalidate or Hit Writeback without Invalidate CACHE suboperations hit on Data Cache and the line data are dirty.
2. A read or write miss occurs and the line data are dirty. In this case the line has to be written to memory before it can be replaced by the miss data.

7.3.5 Data Cache State Transitions

As discussed previously, lines in the Data Cache can be in one of several states: ***Invalid***, ***Valid Clean*** or ***Valid Dirty***.

Invalid means the Data Cache entry does not contain valid data. Upon a miss, the cache can load data into this cache line with no further actions.

The *Valid Clean* state indicates that there are valid data in the Data Cache line and they are the same as memory. All writeback segments have their data in the *Valid Clean* state until they are written to by the processor.

The C790 supports the write-back protocol, hence the need for a *Valid Dirty* state. A Data Cache line transitions to the *Valid Dirty* state when the cache line is written to without reflecting the operation on the bus - the writeback protocol. In this case, the data in the cache does not match the data in memory.

Figure 7-4 shows the transition diagram of the Data Cache performing according to the writeback policy. For details on the CACHE operation, refer to Appendix C.

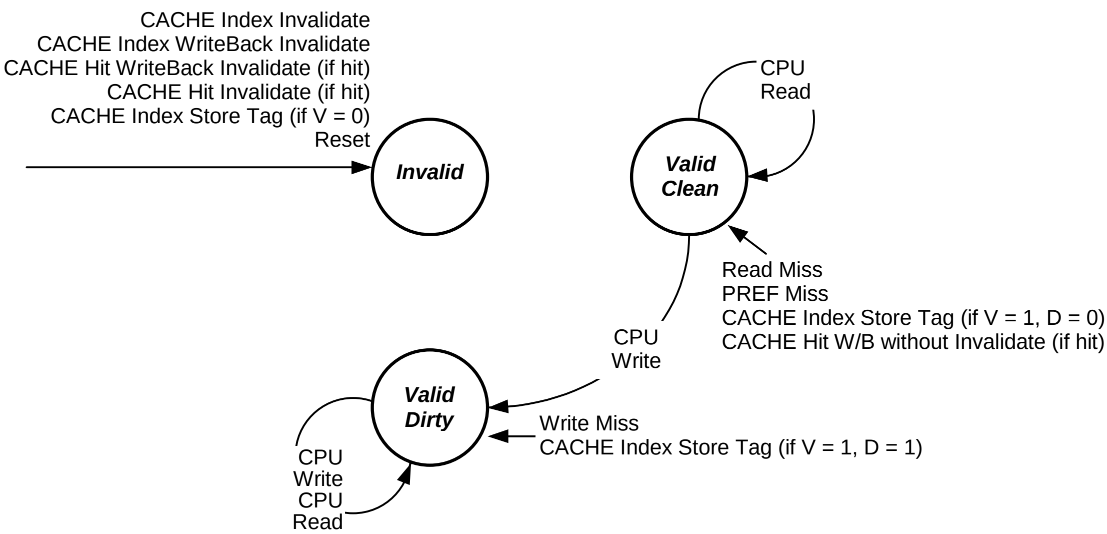


Figure 7-4. Data Cache Transition Diagram, Writeback Protocol

7.3.6 Instruction Cache State Transitions

Cache lines in the Instruction Cache can be in either of two states: ***Invalid*** or ***Valid***.

Invalid means the Instruction Cache entry does not contain valid instruction data. Upon a miss, the cache can load instructions into this cache line with no further actions.

The *Valid* state indicates that there are valid instructions in the cache line and so there is no need for miss processing.

The transition diagram for the Instruction Cache is simple; refer to Figure 7-5. For details on the CACHE instructions refer to Appendix C.

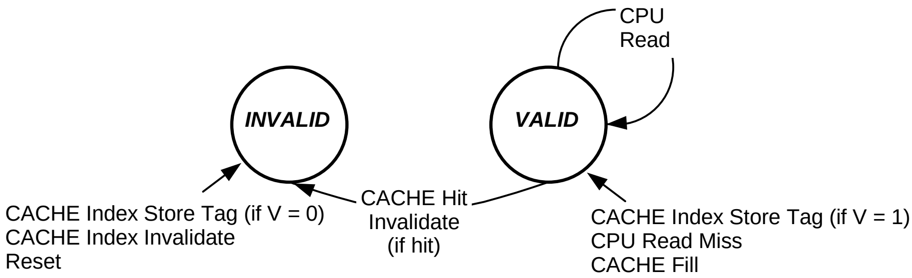


Figure 7-5. Instruction Cache Transition Diagram

7.3.7 Data Cache Lock Function

In a 2-way set-associative Data Cache, such as the one present in the C790, there is no explicit way of forcing data to be retained in the cache. The LRF-based mechanism dynamically determines which cache line should be replaced. A Data Cache lock function has been defined to aid in retaining critical pieces of data in the Data Cache under strict program control.

Each entry on each way of the Data Cache has a Lock (L) bit. The Lock bit aids in locking the line by writing directly into it. After locking the line, the LRF bit is no longer meaningful. Thus, if one of the ways for a particular line is locked, the other way is the only way available for caching. Thus, once a line is locked with a particular physical address tag, any other virtual address which maps onto the same cache line will have only a direct mapped location rather than a 2-way location.

To lock the Data Cache, the following two CACHE instruction suboperations can be used:

INDEX STORE TAG (DCACHE)

INDEX STORE DATA (DCACHE)

For details of the above CACHE instruction suboperation refer to Section 7.6. To lock a Data Cache line, the following code sequence can be used:

```
li       t0,0x00010068 //PTagLo = 0x00010, D=V=L=1, R=0
mtc0     t0,$28        //t0 -> TagLo
sync.l
cache    18,0(r0)      //TagLo -> Tag(way0)
sync.l
la       s0,0x00010000
sw       t1,0(s0)      //store contents of t1 into
                            //locked cache line
```

In this example, the tag has been modified using the CACHE instruction and the data has been updated using a Store instruction.

The following restrictions apply to line locking:

- The result of re-locking a locked line is undefined
- The results of locking both ways of a cache line are undefined

To unlock Data Cache lines, the following code sequence can be used:

```
li          t0,0x00010060    //D=V=1, L=R=0
mtc0        t0,$28           //t0 -> TagLo
sync.l
cache       18,0(r0)         //TagLo -> Tag(way0)
sync.l
```

7.3.7.1 Operations During Lock

When the lock bit is set for cache line (index), only the other way is available for handling cache misses. The misses are blocking. A write access to a locked line in the Data Cache takes place only to the cache without affecting the state of memory. Writes to locked cache lines will ***not*** set the DIRTY (D) bit.

7.3.8 Relationship Between Cached and Uncached Operations

Uncached and Uncached Accelerated load and store operations are always executed in order on the CPU bus. Cached load operations can precede earlier store data present in buffers on the CPU bus. All store data present in buffers prevents a *SYNC* (or *SYNC.L*) instruction from completing until the store data has been sent either to the Data Cache or the CPU bus.

Stores with the uncached and uncached accelerated attributes bypass the Data Cache completely.

7.4 Uncached Accelerated Buffer

The C790 has a small size of read only cache memory for uncached accelerated area to reduce bus traffic. This read only cache, the Uncached Accelerated Buffer (UCAB), can introduce data to itself only by refill process due to a load miss on the UCAB. Once load instructions hit on the UCAB, data are provided directly from the UCAB. The UCAB is invalidated under the following conditions:

- Any load operation which doesn't hit the UCAB, or
- Any store operation, or
- A *SYNC* (or *SYNC.L*) operation, or
- Any exception

Snoop is not supported for the UCAB.

7.4.1 UCAB Configuration

The UCAB is configured as shown in Table 7-6.

Table 7-6. UCAB Configuration

	Size	Organization	Line Size	Refill Size
Uncached Accelerated Buffer	128 bytes	Direct Map	128 bytes	128 bytes

7.4.2 Tag Structure

The UCAB is also indexed by the virtual address, the tag comparison is physical. Table 7-7 shows the UCAB size and access bits.

Table 7-7. UCAB Size and Access Bits

	Size	Way	Size	UCAB Virtual Index Bits	UCAB Tag Size	UCAB Tag Virtual Index Bits
UCAB	128 B	Direct Map	1×128 Bytes	6:4	1×25 Bits	—

The least significant 5 bits of the UCAB Tag ([11:7]) is identical with the virtual address [11:7]. The UCAB Tag has one bit of valid bit. The UCAB Tag doesn't have Ditty, LRF, Lock bits. The valid bit of UCAB Tag is initialized to 0 upon reset.

7.4.3 Non-blocking Loads and HiT under Miss

The UCAB also supports non-blocking load and hit under miss as well as the Data Cache. Non-blocking load and Hit under miss allow the pipeline to continue instruction execution until one of following occurs when an Uncached Accelerated Buffer miss occurs:

1. A subsequent instruction has data dependency with the load that is pending (to be retired).
2. A Data cache miss occurs or a miss occurs on the UCAB.
3. An uncached load instruction is issued.
4. A pipeline0 stalls.

7.5 Cache Control Registers

The operations of the caches are controlled by certain programmable bits in the *Config* register. These bits are:

ICE	Instruction Cache Enable
DCE	Data Cache Enable
IC	Instruction Cache Size
DC	Data Cache Size
IB	Icache Line Size
DB	Dcache Line Size

For details of these configuration bits refer to the COP0 register section.

The two cache tag registers *TagLo* and *TagHi* are 32-bit read/write registers that hold the tag and state of the cache line during initialization and diagnostics. The Tag registers are manipulated by MTC0 and CACHE instructions.

TagLo

31 – 12	11 – 7	6	5	4	3	2 – 0
PTagLo	**0**	**D**	**V**	**R**	**L**	**0**

TagHi

where

PTagLo	Specifies physical address bits 31:12
D	Cache State DIRTY bit (Not used for the Instruction Cache)
V	Cache State VALID bit
R	LRF Bit
L	LOCK Bit (Not used for the Instruction Cache)
0	Must be written as zeros, will return zero on reads

The *TagHi* register contains instruction- and operation-specific items (see the next section).

7.6 CACHE Instruction

For information on the CACHE instruction, please refer to Appendix C.

8. CPU Bus

The C790 CPU core is connected to the rest of the system[1], and to external devices, through the group of on-chip C790 system bus signals called the **CPU Bus**. This chapter defines the architecture of the CPU Bus and describes it in the context of an overall system design.

This chapter describes the following:

- the CPU Bus architecture and agents on the CPU Bus
- the types of transactions possible between agents on the bus
- the bus protocols for transactions

[1] The system consists of a DMA Controller (DMAC) as a master, and various slave devices.

8.1 Introduction

The CPU Bus is an on-chip bus in a highly integrated processor. All **agents** (see definitions section 8.1.1 below) on the CPU Bus are equipped with a CPU Bus interface unit connected via CPU Bus signals. An agent acts like a master when it initiates reads or writes on the bus. An agent acts like a slave when it responds to reads or writes initiated by a master. For the CPU Bus to operate properly, an arbiter is needed, to perform arbitration between the CPU and the other bus masters. The arbiter is located in the CPU, and CPU arbitration behavior is discussed in Section 8.5.1, Arbitration Operations.

The following are main features of the CPU Bus:

- Separate data and address buses (Demultiplexed operation)
- 128-bit data bus
- Clocked synchronous operations
- Peak transfer rate of 2.1GB/sec (@133 MHz bus clock)
- 8/16/32/64/128-bit and burst accesses
- Multimaster capability
- Pipelined operations
- No turn-around or dead cycles between transfers

The CPU Bus does not provide:

- Cache coherency support
- Split transactions

8.1.1 Terminology

Address Phase is the cycles during which an address is driven on the CPU Bus through the cycle the address is acknowledged.

Agent refers to different devices on the CPU Bus.

Assert means taking a signal to its active level. An active high signal is "1" when asserted, and an active low signal is "0" when asserted.

CPU means the C790 CPU. The terms CPU and C790 are used interchangeably in this chapter.

Data Phase is the cycles during which data are driven on the bus through the cycle they are acknowledged.

DMAC is the DMA Controller in the system.

Master means the current bus master on the CPU Bus.

MEM refers to the system memory controller.

Negate/Deassert means taking a signal to its inactive state. An active high signal is "0" when deasserted. An active low signal is "1" when negated.

* (after signal name) means active low signal.

8.1.2 Signal Naming Convention

Table 8-1 shows the prefixes used for naming signals in a system incorporating the C790 CPU Bus.

Table 8-1. System Signal Naming Convention

Signal Prefix	Signal Type
CPU	Signals from the CPU multiplexed or logically combined with the DMAC signals to form the system signals. These signals include: CPUADDR, CPUBE*, CPURD*, CPUWR*, CPUTSIZE, CPUASTART*, CPUDSTART*, CPUDATA.
SYS	The combined or multiplexed signals from any agents on the CPU Bus. These signals include: SYSADDR, SYSBE*, SYSRD*, SYSWR*, SYSTSIZE, SYSASTART*, SYSDSTART*, SYSAACK*, SYSDACK*, SYSDATA.

8.2 CPU Bus Architecture

The CPU Bus design is a synchronous pipelined bus with separate data (128-bit) and address buses running at half the clock frequency of the CPU. The CPU is connected to the rest of the system and external devices through this bus. Figure 8-1 illustrates the architecture of the bus and identifies different agents that can be on the bus.

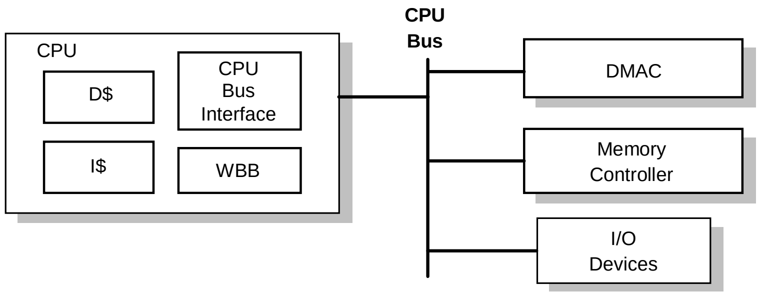


Figure 8-1. CPU Bus Architecture

8.2.1 CPU Bus Connectivity for Address and Control Paths

Figure 8-2 illustrates the system-level interconnections for address paths of the CPU Bus.

Support logic is needed to handle the fact that the system contains multiple masters. AGNT* is used to control the multiplexer in the support logic that selects a master to be connected to the CPU Bus.

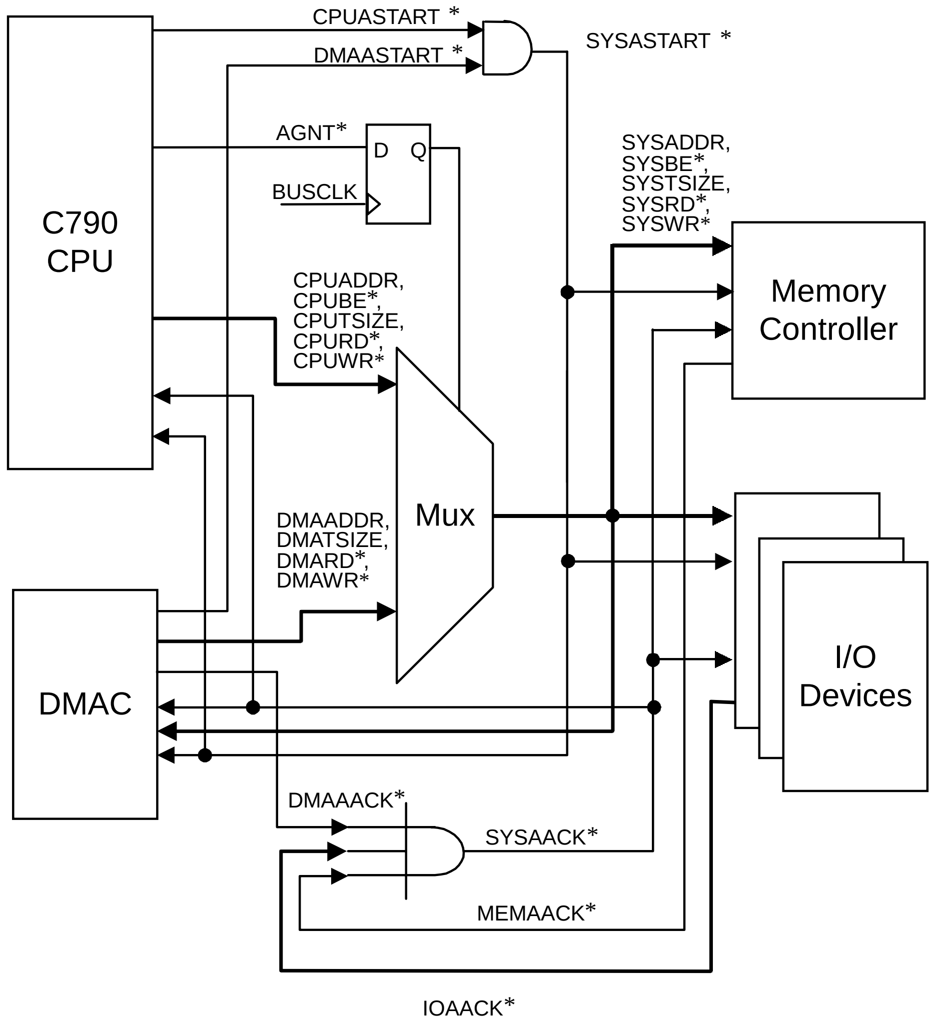


Figure 8-2. CPU Bus Address and Control Path Connections in System

8.2.2 CPU Bus Connectivity for Data Paths

Figure 8-3 illustrates the system-level interconnections for data paths of the CPU Bus.

For read cycles, the support logic must control the multiplexer so that the correct source of data is put on SYSDATA.

For write cycles, the support logic must detect whether the cycle is a CPU cycle or a DMA cycle, and use this to control the multiplexer.

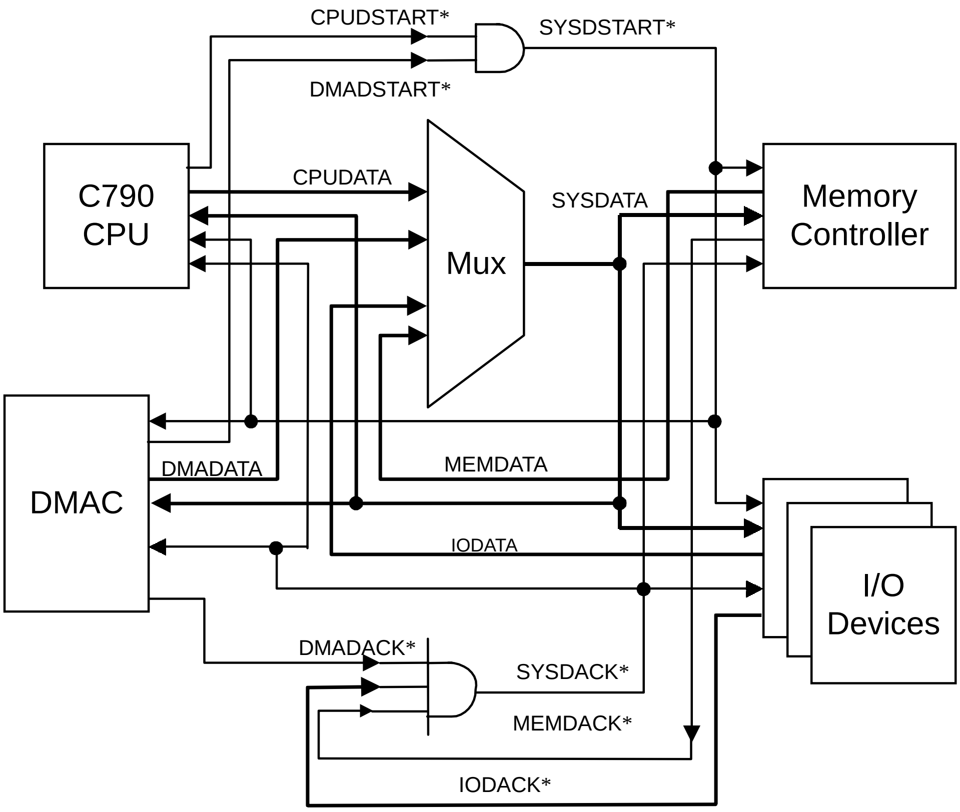


Figure 8-3. CPU Bus Data Path Connections in System

8.3 CPU Bus Signal Descriptions

This section describes the CPU Bus signals and their usage in different bus operations.

8.3.1 Address Bus Signals

CPUADDR[31:4] CPU address bus

CPUADDR[31:4] bits are valid during the address phase and can be sampled by the slave when CPUASTART* is sampled low.

SYSADDR[31:4] System address bus

SYSADDR[31:4] are multiplexed outputs selecting between CPUADDR[31:4] and DMA address. They are valid during the address phase and can be sampled by the slave when SYSASTART* is sampled low.

CPUBE[15:0]* CPU byte enables

CPUBE[**i**]*, driven during the address phase, indicates valid data on byte **i** of CPUDATA[127:0] during the data phase. CPU byte enables can be sampled by the slave when CPUASTART* is sampled low. CPU byte enables are used only in CPU single cycles.

SYSBE[15:0]* System byte enables

SYSBE[**i**]*, driven during the address phase, indicates valid data on byte **i** of SYSDATA[127:0] during the data phase. System byte enables can be sampled by the slave when SYSASTART* is sampled low. System byte enables are used only in CPU single cycles.

CPUTRANSTYPE[4:0] CPU transaction type

CPUTRANSTYPE[4:0], driven during the address phase, indicates the type of operation. CPU transaction type can be sampled by the slave when CPUASTART* is sampled low.

Table 8-2. Bus Transaction Types

CPUTRANSTYPE	Type of Bus Transaction
00000	Not defined or miscellaneous
00001 - 00111	Reserved
01000	Data Cache Refill due to Load Miss
01001	Data Cache Refill due to Prefetch Instruction
01010	Data Cache Refill due to Store Miss
01011	Uncached Load
01100	Uncached Accelerated Load
01101 - 01111	Reserved
10000	Instruction Cache Miss Refill
10001	Cache Instruction - Fill Suboperation
10010	Uncached Execution
10011 - 10111	Reserved
11000	Data Cache Write-back due to Load/Store Miss
11001	Data Cache Write-back due to Cache Instruction
11010	Uncached Store
11011	Uncached Accelerated Store
11100	Non-allocated Store
11101 - 11111	Reserved

CPURD* CPU read

The CPU asserts this signal to indicate a read operation. This signal can be sampled when CPUASTART* is sampled low. This signal is active during the address phase. CPURD* is used in transfers initiated by the CPU.

CPUWR* CPU write

The CPU asserts this signal to indicate a write operation. This signal can be sampled when CPUASTART* is sampled low. This signal is active during the address phase. CPUWR* is used in transfers initiated by the CPU.

CPUTSIZE[1:0] — CPU transfer size

While driven by the CPU, these signals indicate the size of the transfer in the current CPU initiated bus cycle. They are driven during the address phase and can be sampled starting at the edge where CPUASTART* is sampled low.

Table 8-3. CPU Transfer Size

CPUTSIZE[1:0]	Transfer Size
00	1 Quadword (Single Cycle)
11	4 Quadwords

SYSTSIZE[2:0] — System transfer size

While driven by the system, these signals indicate the size of the transfer in the current system bus cycle. They are driven during the address phase and can be sampled starting at the edge where SYSASTART* is sampled low.

CPUASTART* — CPU address start

Driven by the CPU, it indicates the start of the address phase. Address, byte enable, and control signals (CPUADDR[31:4], CPUBE[15:0]*, CPURD*, CPUWR*, and CPUTSIZE) can be sampled to determine the type of cycle requested starting where CPUASTART* is sampled low. CPUASTART* is driven active for only one cycle.

SYSASTART* — System address start

SYSASTART* is driven by the system; it indicates the start of the address phase. Address, byte enable, and control signals can be sampled to determine the type of cycle requested starting where SYSASTART* is sampled low. SYSASTART* is driven active for only one cycle.

SYSAACK* — System address acknowledge

This signal is an input to all the agents on the CPU Bus indicating that address and control signals have been sampled by the slave. The master terminates the address phase one cycle after sampling SYSAACK* low.

CPUDATA[127:0] — CPU data bus

This is a 128-bit data bus output from the CPU.

SYSDATA[127:0] — System data bus

This is the 128-bit data bus input to all devices on the CPU Bus.

CPUDSTART* CPU data start

During read/write operations, this output from the CPU indicates the start of data phase. For CPU write operations, the slave can sample data from the bus one cycle after CPUDSTART* has been asserted. For CPU read operations, the slave can output data on the bus any cycle after the cycle CPUDSTART* has been asserted.

SYSDSTART* System data start

During read/write operations, this output from the system indicates the start of data phase. Data transfer can begin one cycle after SYSDSTART* has been asserted. For DMA cycles, if the slave, providing the data, cannot supply data in the next cycle after the assertion of SYSDSTART*, it is the responsibility of the designer to come up with a new DMA protocol.

SYSDACK* System data acknowledge

This signal is an input to all the agents on the bus indicating the valid status of data on the bus. During read cycles, it indicates read data are available on the bus to be sampled by the master. During write cycles, it indicates the slave has sampled the data. This signal should be asserted for each data transfer during burst operations. During read transactions, data are sampled one cycle after SYSDACK* has been asserted. During write transactions, the master drives new data on the bus one cycle after detecting SYSDACK* low.

BUSERR* Bus error

This signal is an input to the CPU and the DMAC which indicates that a bus error has occurred during the transaction. BUSERR* serves to terminate the bus protocol and return bus ownership to the CPU.

INT[1:0]* Interrupt request lines

These signals are interrupt inputs to the CPU.

SIOINT* Serial I/O interrupt request

This line provides the serial I/O interrupt from the I/O controller.

NMI* Non-maskable interrupt

Non-maskable interrupt input to the CPU.

SYSBIGENDIAN Big Endian enable

This input signal is sampled during cold reset and make CPU to operate as big endian when it is asserted. The input level of this signal must not be changed during the operation.

CPCOND0 Coprocessor conditions

These lines are an input to the CPU as test conditions for some of the branch instructions.

RESET* Reset

Input to the CPU. When this line is asserted, the CPU, DMAC and slave devices execute a reset.

CPUCLK CPU clock

CPU clock

BUSCLK Bus clock

Bus clock: 1/2, 1/3 or 1/4 frequency of the CPUCLK.

AREQ* Address bus request

This signal is an output from the DMAC to the CPU. When it is asserted, the DMAC requests the address bus mastership.

AGNT* Address bus grant

This signal is an output from the CPU to grant the bus mastership to the DMAC. This signal is asserted in response to assertion of the AREQ* signal.

REL* Bus release request

This signal is asserted by the CPU to request that the current bus owner release the CPU Bus.

8.4 Overview of CPU Bus Operations

This section discusses CPU Bus operations; it covers processor requests, DMA operations, and bus error operation.

In this section descriptions show CPU signals followed by the system lines, in parentheses, onto which they are asserted. For example: CPUASTART* (SYSASTART*) means CPUASTART* is asserted on the SYSASTART* line. Where a value is given, the bits output by the CPU are shown, followed by the bits, in parentheses, on the system lines. For example if we have 11 on CPUTSIZE[1:0], during a CPU bus cycle, then we will get 011 on the SYSTSIZE[2:0]. This will be shown as 11 (011).

8.4.1 CPU Bus Operations

The CPU Bus is different from conventional buses in that it allows ***pipeline*** operations. In this case, pipeline implies up to two outstanding requests before any data transaction has taken place. For instance, the CPU may issue two back-to-back read requests to main memory before any data have been returned. Note that at any time, there can only be two outstanding requests on the bus. The master requiring more than two operations has to wait until the first request has been serviced completely prior to issuing the third one.

8.4.2 Processor Requests

The CPU issues single requests, burst requests or a series of requests to other agents on the bus. These requests are referred to as *processor requests* initiated through the CPU Bus interface.

The processor requests are in response to the following system events:

- Load miss
- Store miss
- Write-back buffer writes (dirty data cache lines, uncached writes, etc.)
- Uncached loads and uncached accelerated loads
- Instruction miss and uncached instruction fetch

Processor read/write requests can be a burst, quadword, or partial quadword of data to and from the main memory or any other system resources. A processor-initiated burst is always 4 quadwords.

8.4.2.1 Read Requests

The CPU initiates read requests by driving address and control on the bus and asserting CPUASTART* (SYSASTART*) to indicate valid address and control. The CPU will keep driving address and control until the slave device has acknowledged the address phase by asserting address acknowledge, SYSAACK*. For burst reads, the CPU drives CPUTSIZE (SYSTSIZE) to 11 (011) to indicate burst reads. The CPU also indicates that it is ready to accept read data by asserting CPUDSTART* (SYSDSTART*). The slave device returns the requested data on the data bus by asserting SYSDACK*, data acknowledge.

8.4.2.2 Write Requests

The CPU initiates write requests by driving address and control on the bus and asserting CPUASTART* (SYSASTART*). The CPU also drives data on the bus and indicates that by asserting CPUDSTART* (SYSDSTART*). The slave device accepts the address and data by asserting SYSAACK* and SYSDACK*, respectively. Burst writes are indicated by driving CPUTSIZE (SYSTSIZE) to 11 (011) during the address phase.

8.4.3 Bus Error Operations

Bus error occurs when the CPU or DMA initiates cycles but there are no devices on the CPU Bus responding to the cycles. The absence of response to either the address phase or the data phase will cause the bus error condition. The bus error is always imprecise.

When bus error occurs, all the agents including the CPU, DMAC, and slave devices on the CPU Bus will terminate the current bus cycle.

In the case where CPU is the initiator of the cycle, there can be two types of bus error:

- Data load/store bus error
- Instruction fetch bus error

Bus error sets the corresponding exception bit in the *CAUSE* register. Subsequently, the CPU will jump to the proper error handler for the examination of the exception. However, the bus error exception is imprecise. There is no guarantee that the CPU can recover from this error condition.

In case the DMAC is the initiator of the cycle, the types of bus error depends on the implementation of the DMAC. After bus error occurs, the DMAC will release the bus mastership back to the CPU and assert interrupt or NMI to the CPU. The interrupt or NMI routine will then handle the bus error condition for the DMAC.

8.5 CPU Bus Transaction Protocols and Timing

This section describes transaction protocols and the timing for the following CPU Bus operations:

- Arbitration
- CPU single operations (one quadword)
- CPU burst operations (four quadwords)
- CPU non-pipelined single operations (one quadword)
- CPU non-pipelined burst operations (four quadwords)
- Bus error operations

8.5.1 Arbitration Operations

An arbiter is required to mediate between devices requesting the CPU Bus. The arbiter is located in the CPU. The CPU is the **default** bus master; AREQ* and AGNT* are both deasserted during RESET.

A master other than the CPU may request the bus by asserting the request signal, AREQ*. In response to the AREQ* signal, the CPU will issue the grant signal, AGNT*, to grant the address bus to the requesting master. In the cycle AGNT* is sampled active by the bus master, the master starts the address phases and deasserts AREQ* in the beginning of the last address phase. When the corresponding data phases commences, the CPU or the requesting master starts the data transfers depending on the DMA transfer. Data phases follow the exact order of address phases. The arbitration signals are shown in Figure 8-4.

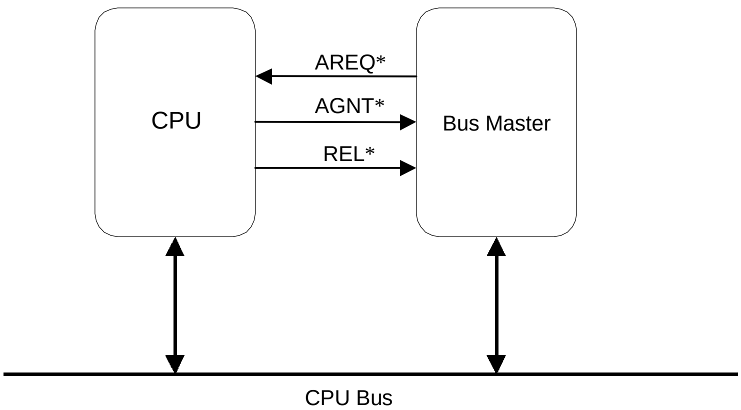


Figure 8-4. Connection of Arbitration Signals

The arbitration priority in using the CPU Bus is that the DMAC always has higher priority than the CPU. When both the CPU and the DMAC arbitrate for the CPU Bus, the arbiter grants the bus mastership to the DMAC. The CPU can assert REL* to the DMAC in an effort to get the bus ownership back from the DMAC. The CPU will proceed with the transfer once the DMAC has released the CPU Bus.

The arbitration cycles and protocol are shown in Figure 8-5. In response to the DMAC asserting its request AREQ*, the arbiter asserts AGNT* in cycle 3 which is the arbitration cycle. The DMAC samples AGNT* asserted and begins its address phases. When the DMAC asserts to begin the last address phase, it deasserts its request line AREQ* in cycle 4. The arbiter then waits for the SYSAACK* cycle to deassert AGNT* to release bus mastership back to the CPU.

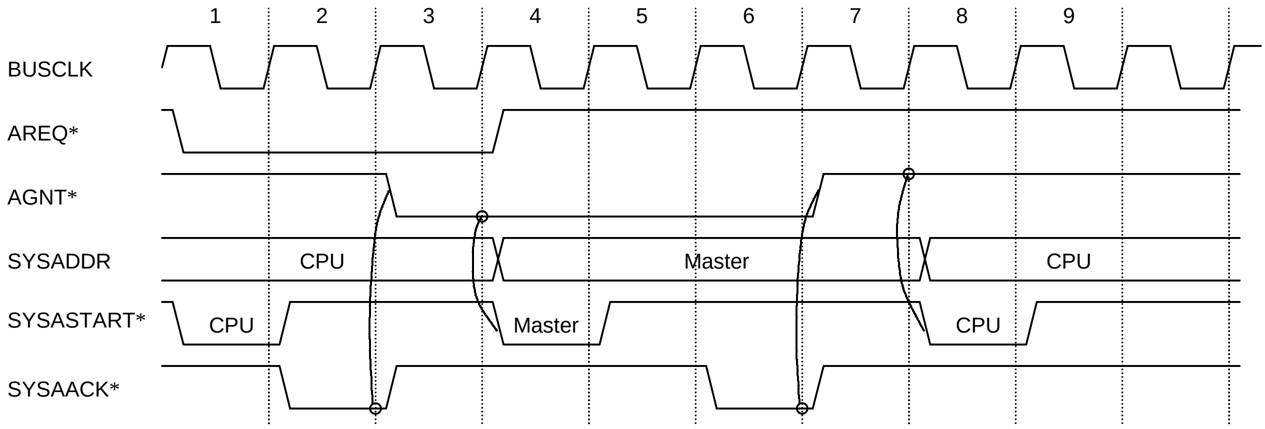


Figure 8-5. Arbitration Protocol

8.5.1.1 Cycle Stealing

Cycle stealing refers to the CPU's ability to preempt a master in order to perform a bus operation. This operation could be either due to the write back buffer (WBB) being almost full (having more than 64 bytes filled up) or the CPU needing to perform an instruction or data read. These operations are collectively referred to as cycle stealing operations.

Figure 8-6 illustrates the cycle stealing protocol. The arbiter asserts the REL* (Release) signal in response to the CPU's request cycles. The master deasserts its request after having finished its operations. When the master has begun the last address phase with the master deasserts the AREQ* signal indicating to the arbiter that the bus will be relinquished; as indicated in cycle 9. When the address phase ends, the address bus is returned to the CPU by the deassertion of AGNT* in cycle 12. The arbiter deasserts REL* at the same time AGNT* is deasserted. The data phases follow the same order as the address phases.

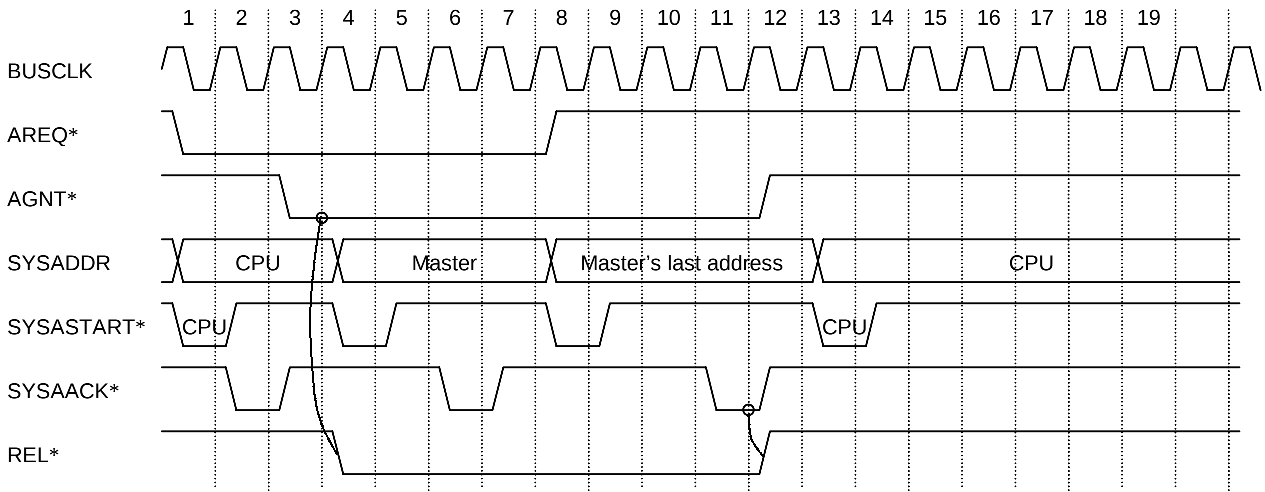


Figure 8-6. Cycle Stealing Protocol

8.5.2 CPU Single Operations

CPU Single operations transfer one quadword.

In single operations, the CPU drives the address, byte enables, and the read/write signals and indicates their valid status by asserting CPUASTART* (SYSASTART*). The slave samples valid address and control lines and responds by asserting SYSAACK*. In single operations, CPUTSIZE (SYSTSIZE) is always 00 (000).

When the CPU detects SYSAACK* active and is ready to put another address on the bus, it will start another address phase. The bus only supports two levels of address pipelining. That means only two address phases can be outstanding before any data phase begins.

The CPU indicates that it is ready to accept/supply data by asserting CPUDSTART* (SYSDSTART*) one cycle prior to actually accepting/supplying it. For read cycles, the slave supplies the data and indicates that the data is ready by asserting SYSDACK*. For write cycles, the CPU supplies data one cycle after CPUDSTART* (SYSDSTART*) is asserted, and the slave accepts the data by asserting SYSDACK*.

8.5.2.1 CPU Single Reads

The fastest CPU single read is 2 cycles. Address and data phases for AddrA illustrate the fastest CPU single read cycle. The CPU asserts CPUASTART* (SYSASTART*) to begin the address phase in cycle 1. The slave device asserts SYSAACK* in cycle 1 to indicate that it has sampled the address. The CPU then begin another address phase in cycle 3. The assertion of SYSDACK* by the slave device in cycle 1 triggers the CPU to sample SYSDATA at the end of cycle 2.

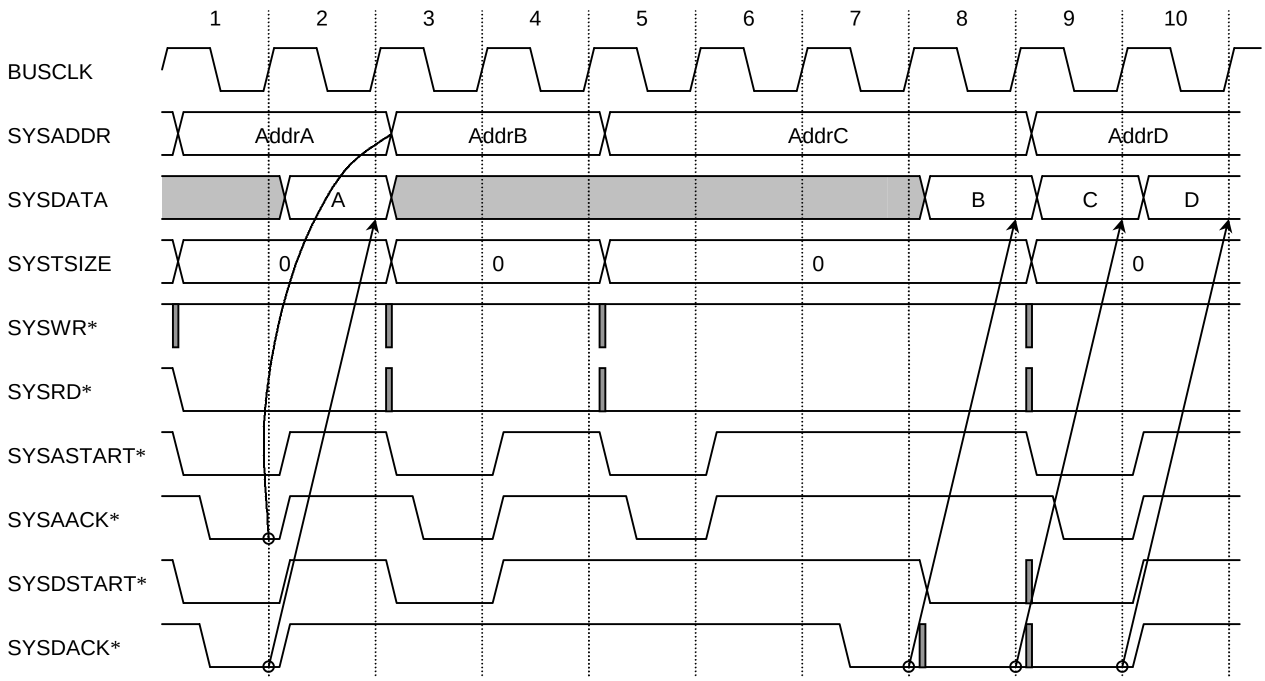


Figure 8-7. CPU Single Reads

8.5.2.2 CPU Single Writes

The fastest CPU single write is 2 cycles. Address and data phases for AddrA illustrate the fastest CPU single write cycle. The CPU always drives data onto CPUDATA one cycle after the assertion of CPUDSTART* (SYSDSTART*). For example, in, the CPU drives CPUDATA in cycle 2 which is one cycle after the assertion of CPUDSTART* (SYSDSTART*) in cycle 1. The slave device samples SYSDATA one cycle after the assertion of SYSDACK*.

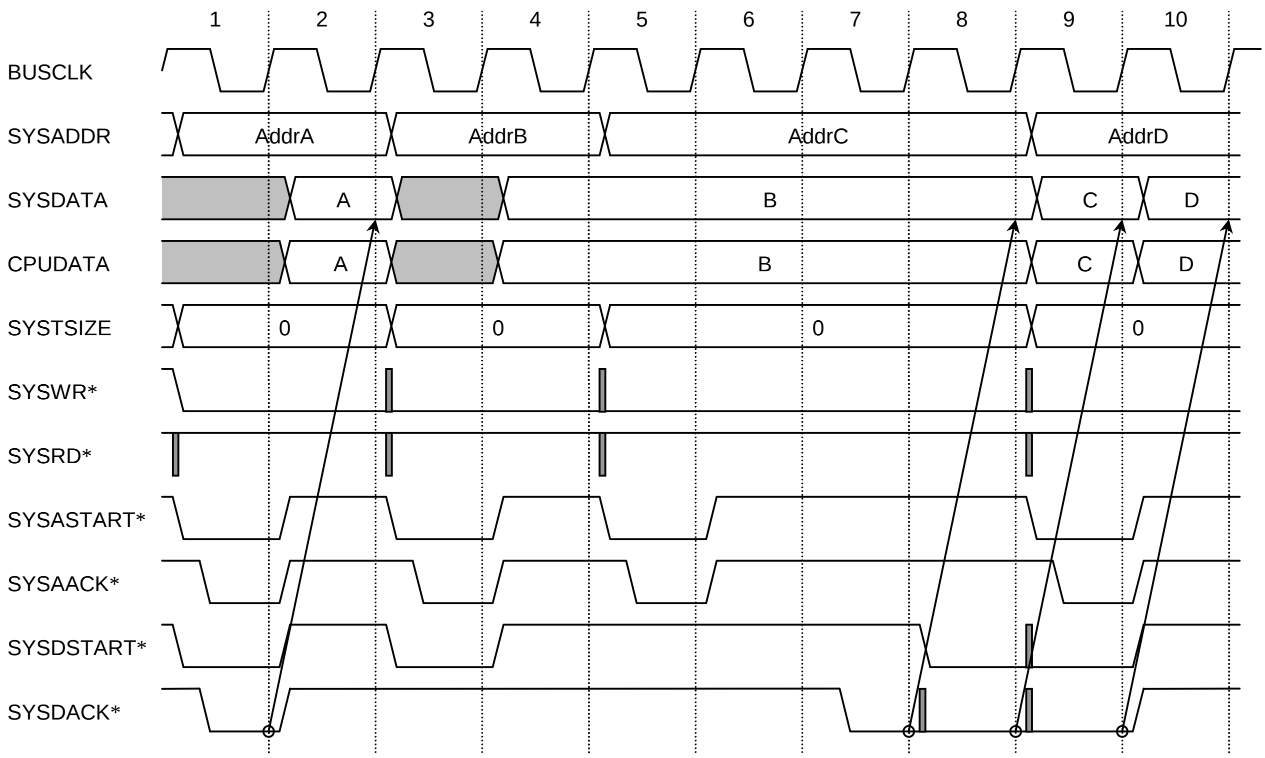


Figure 8-8. CPU Single Writes

8.5.2.3 CPU Single Read-Write-Read-Write Cycles

All adjacent address phases are read-write or write-read cycles. AddrA is a read address and AddrB is a write address, and so on.

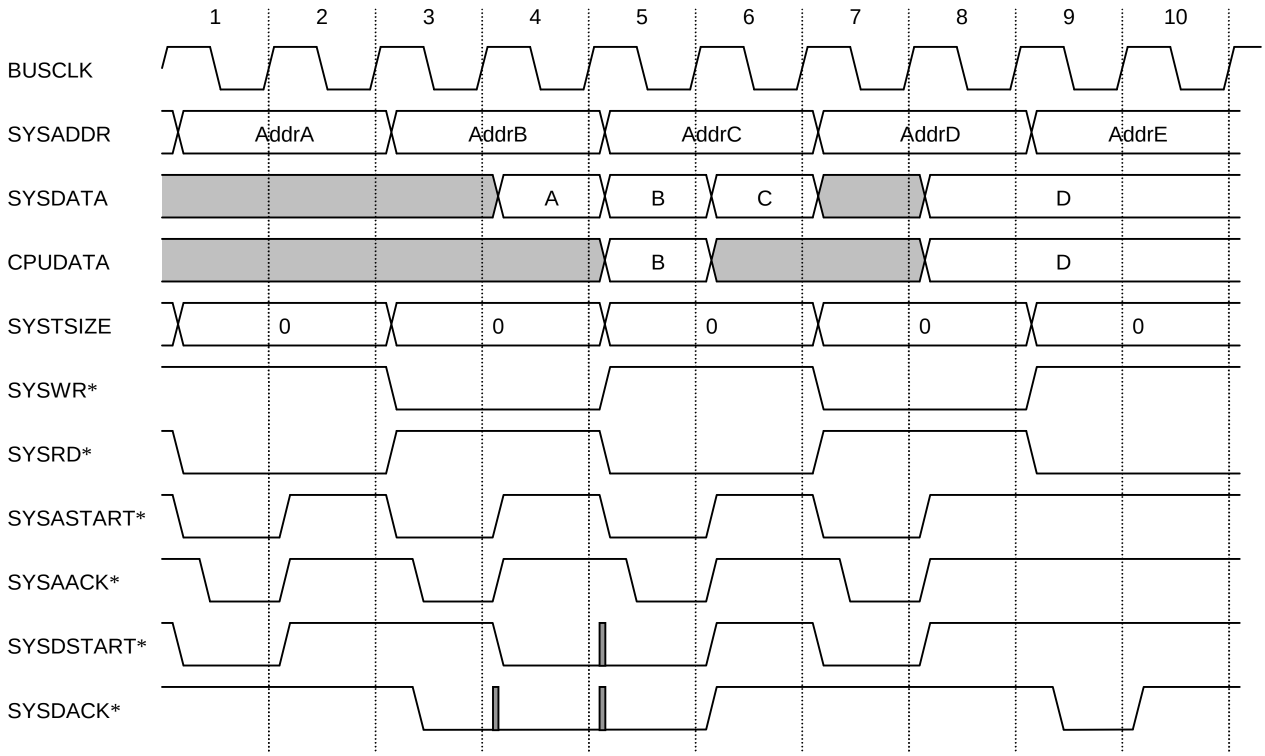


Figure 8-9. CPU Single Read-Write-Read-Write Cycles

8.5.3 CPU Burst Operations

CPU Burst operations transfer four quadwords. In burst operations, the CPU drives the address and control signals and indicates their validity by asserting CPUASTART* (SYSASTART*). The slave samples valid address and control lines and asserts SYSAACK* to acknowledge the address phase. The address phase is the cycles from CPUASTART* (SYSASTART*) asserted to one cycle after SYSAACK* is asserted.

When the CPU detects SYSAACK* active and has another address ready, it will start another address phase.

The CPU indicates that it is ready to accept/supply data by asserting CPUDSTART* (SYSDSTART*) one cycle prior to actually accepting/supplying it. For read cycles, the slave supplies the data and indicates that data are valid by asserting SYSDACK* one cycle prior to the data being available. For write cycles, the CPU supplies data one cycle after CPUDSTART* (SYSDSTART*) is asserted, and the slave accepts the data by asserting SYSDACK*. For burst cycles, there are many SYSDACK* for data transfer.

The CPUTSIZE (SYSTSIZE) indicates the number of quadwords in the transfer. The CPU initiated cycles use only values of either 00 (for CPU Single operations) or 11 (for CPU Burst operations), which are single and burst of 4 quadwords respectively.

8.5.3.1 CPU Burst Reads

The fastest CPU burst read is 5 cycles. Address and data phases for AddrA illustrate the fastest CPU burst read cycle. There are four SYSDACK* sent by the slave device for every CPU burst read cycle. The slave device asserts SYSDACK* in cycle 1, 2, 3, and 4 to indicate that data can be sampled at the end of cycle 2, 3, 4, and 5 by the CPU.

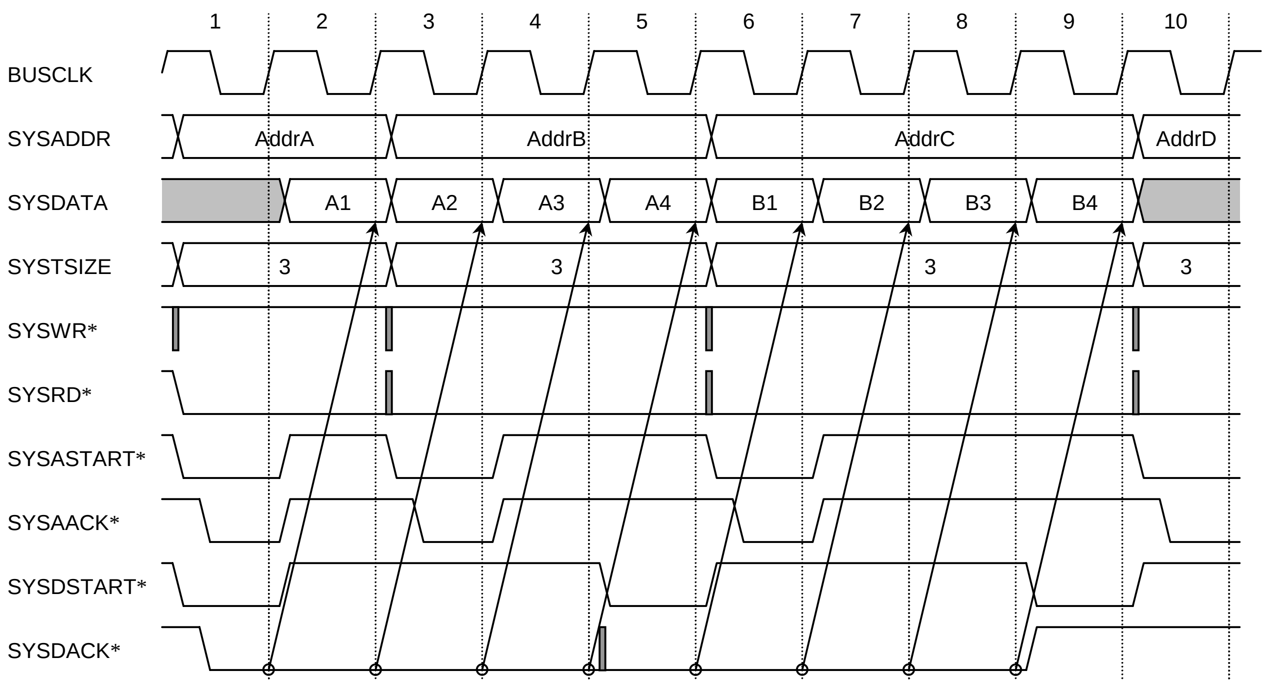


Figure 8-10. CPU Burst Reads

8.5.3.2 CPU Burst Writes

The fastest CPU burst write is 5 cycles. Address and data phases for AddrA illustrate the fastest CPU burst write cycle. After assertion of CPUDSTART* (SYSDSTART*) in cycle 1, the CPU drives the first data on CPUDATA in cycle 2. As SYSDACK* is sampled asserted in cycles 1, 2, 3, and 4, the CPU drives a new data on CPUDATA at the end of cycles 2, 3, 4, and 5.

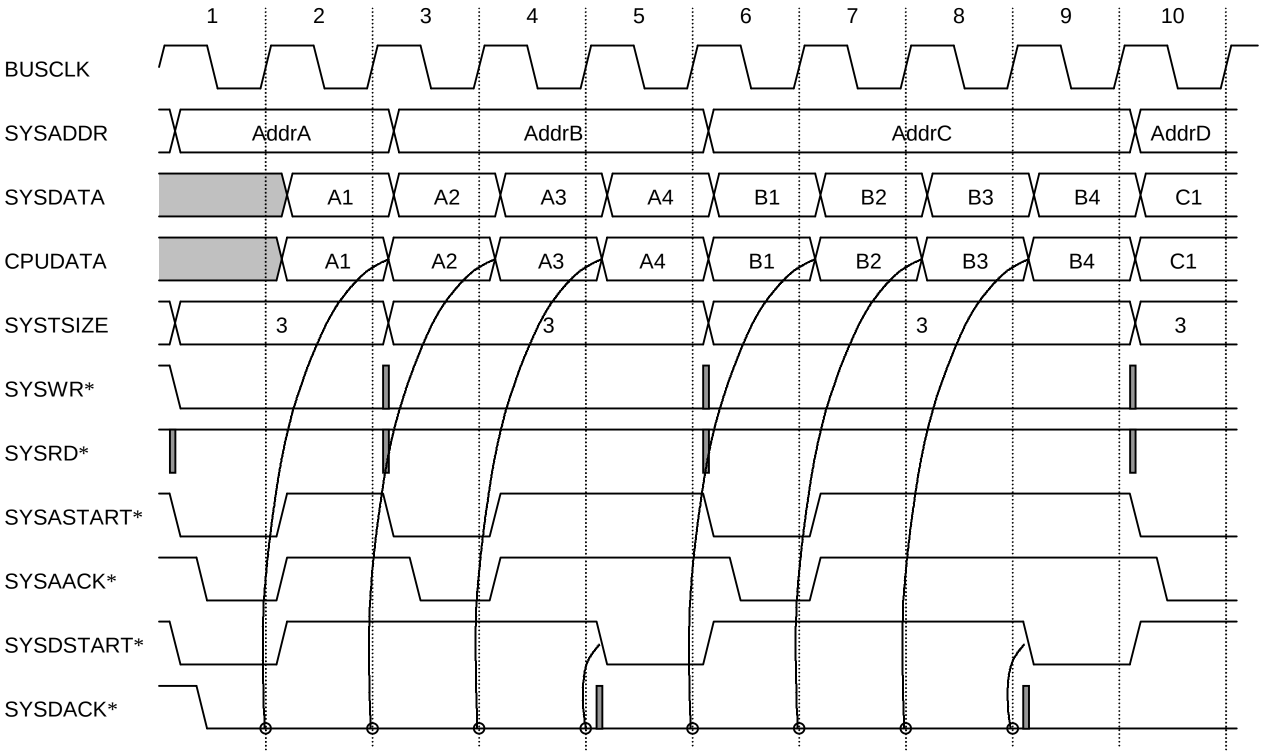


Figure 8-11. CPU Burst Writes

8.5.3.3 CPU Burst Read-Write Cycles

All adjacent address phases are read-write or write-read cycles. AddrA is a read address and AddrB is a write address, and so on.

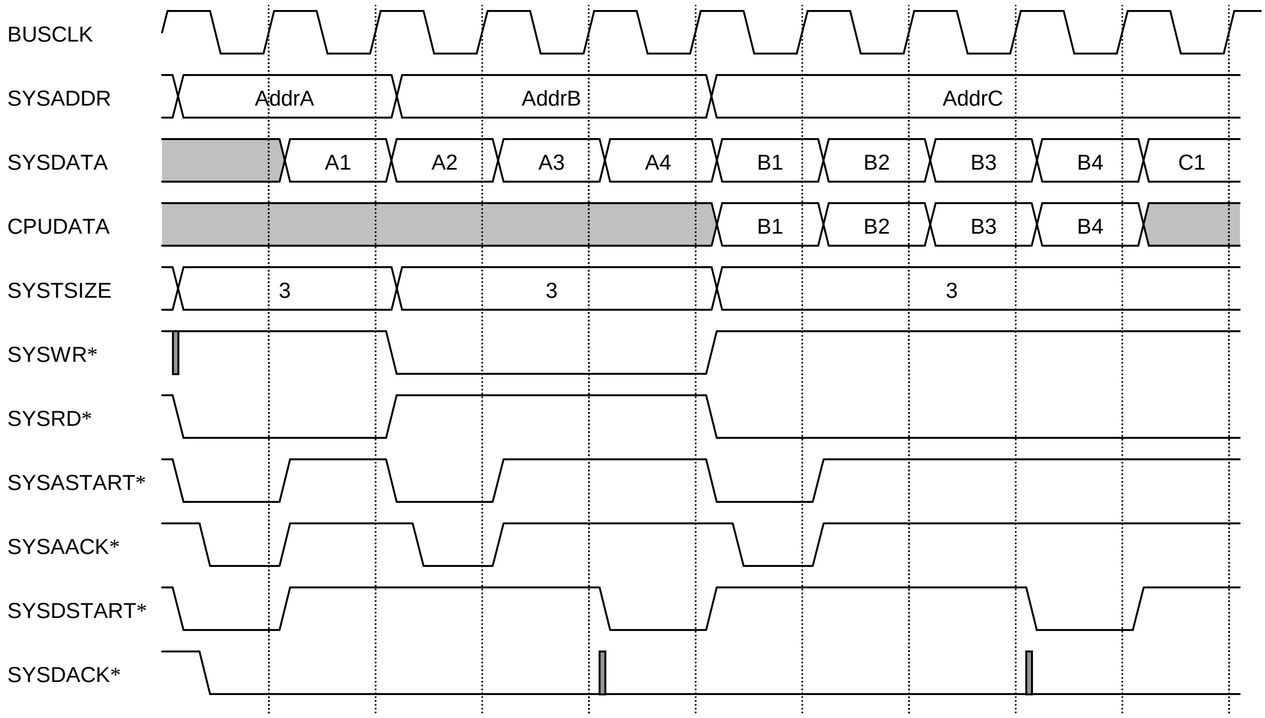


Figure 8-12. CPU Burst Read-Write Cycles

8.5.3.4 CPU Burst Write-Read Cycles

All adjacent address phases are read-write or write-read cycles. AddrA is a write address and AddrB is a read address, and so on.

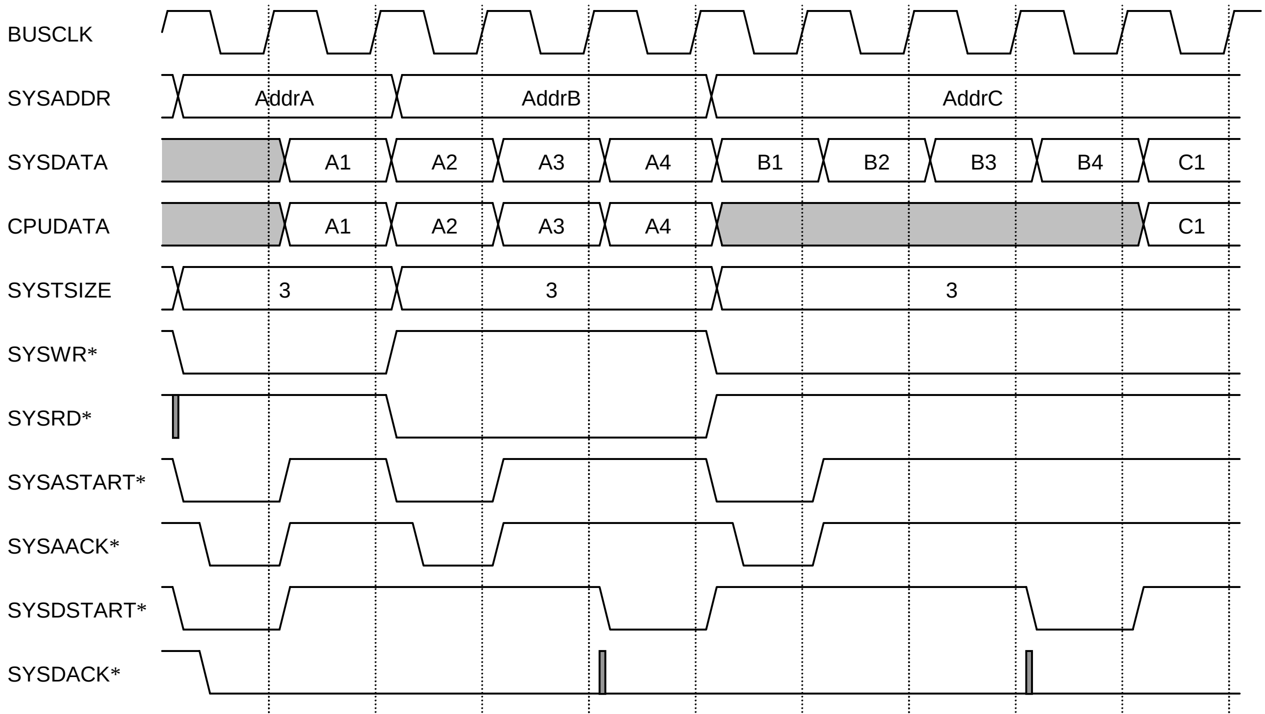


Figure 8-13. CPU Burst Write-Read Cycles

8.5.4 CPU Non-Pipeline Single Operations

The CPU Bus can support non-pipeline operations as well as pipeline operations. The non-pipeline operations are done simply by delaying the assertion of SYSAACK* until the last SYSDACK* of the bus transaction. The advantage of this is that the peripheral does not need to save the current address; it just decodes the address on the address bus for the current operation. Using this mode of operation simplifies the peripheral interfaces to the CPU Bus but it degrades the system performance.

8.5.4.1 CPU Non-Pipeline Single Reads

All adjacent address phases are read cycles.

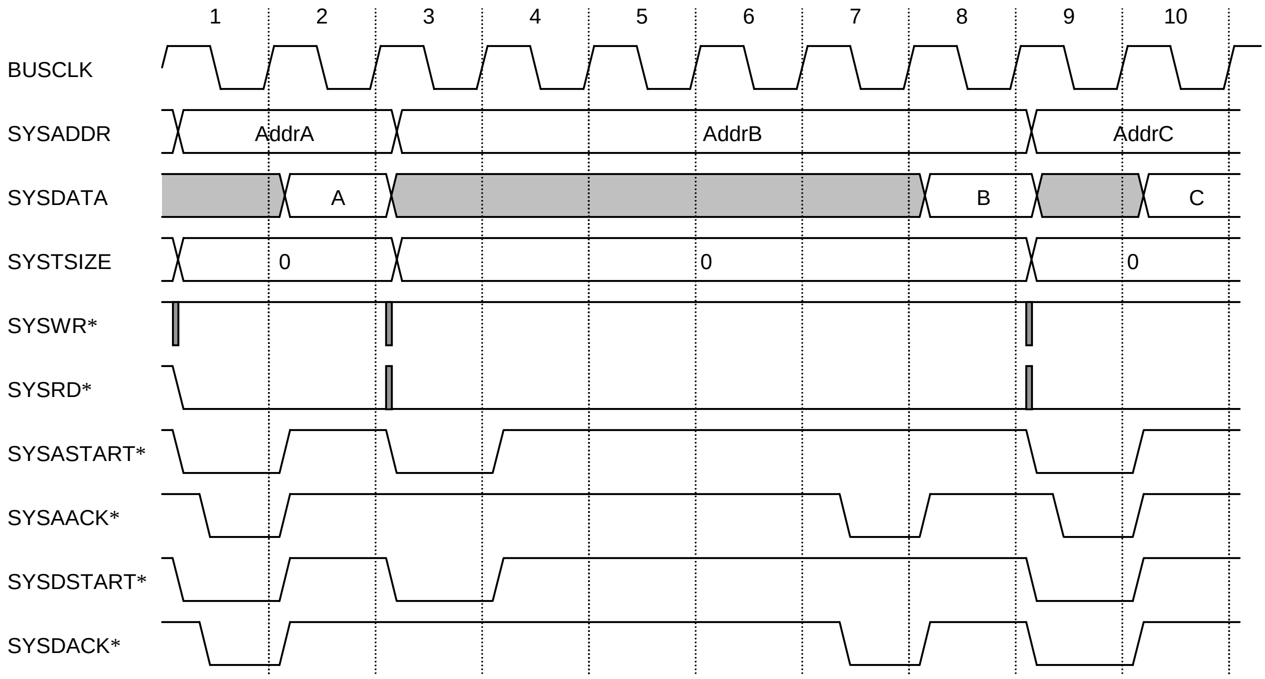


Figure 8-14. CPU Non-Pipeline Single Reads

8.5.4.2 CPU Non-Pipeline Single Writes

All adjacent address phases are write cycles.

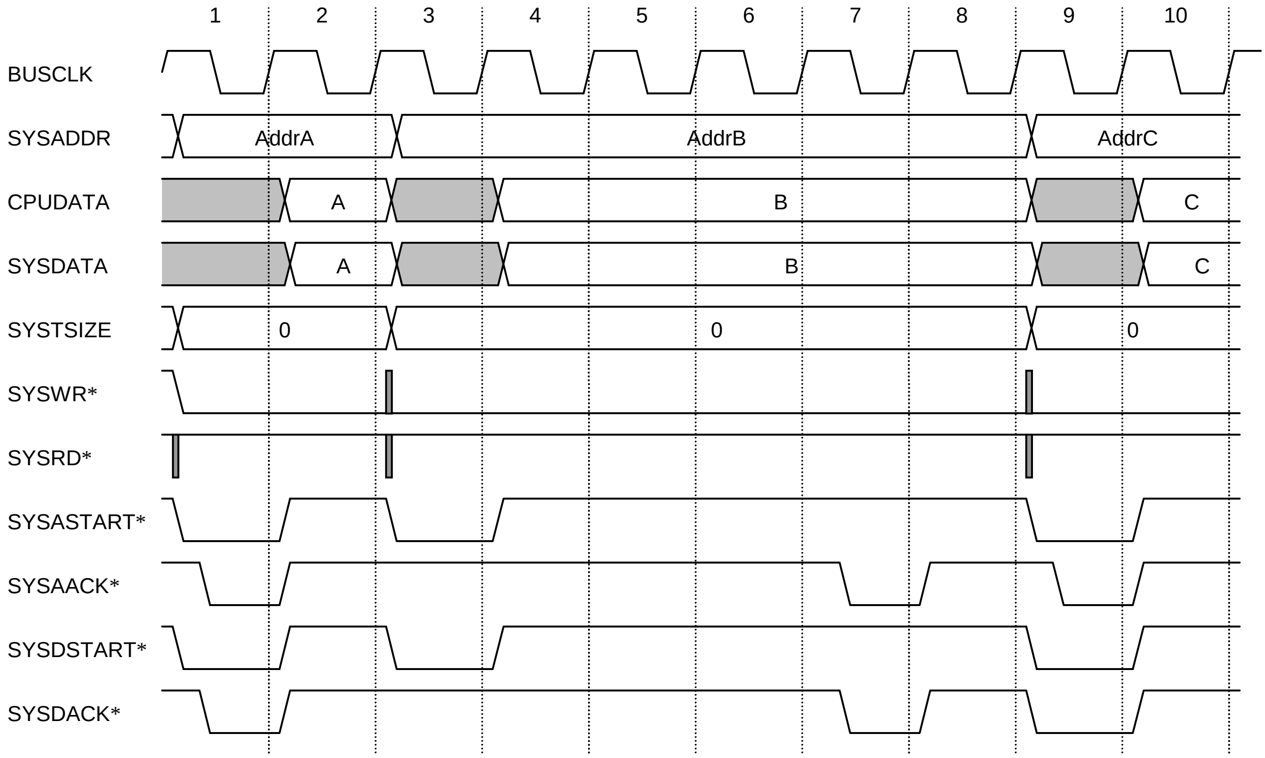


Figure 8-15. CPU Non-Pipeline Single Writes

8.5.5 CPU Non-Pipeline Burst Operations

8.5.5.1 CPU Non-Pipeline Burst Reads

All adjacent address phases are read cycles.

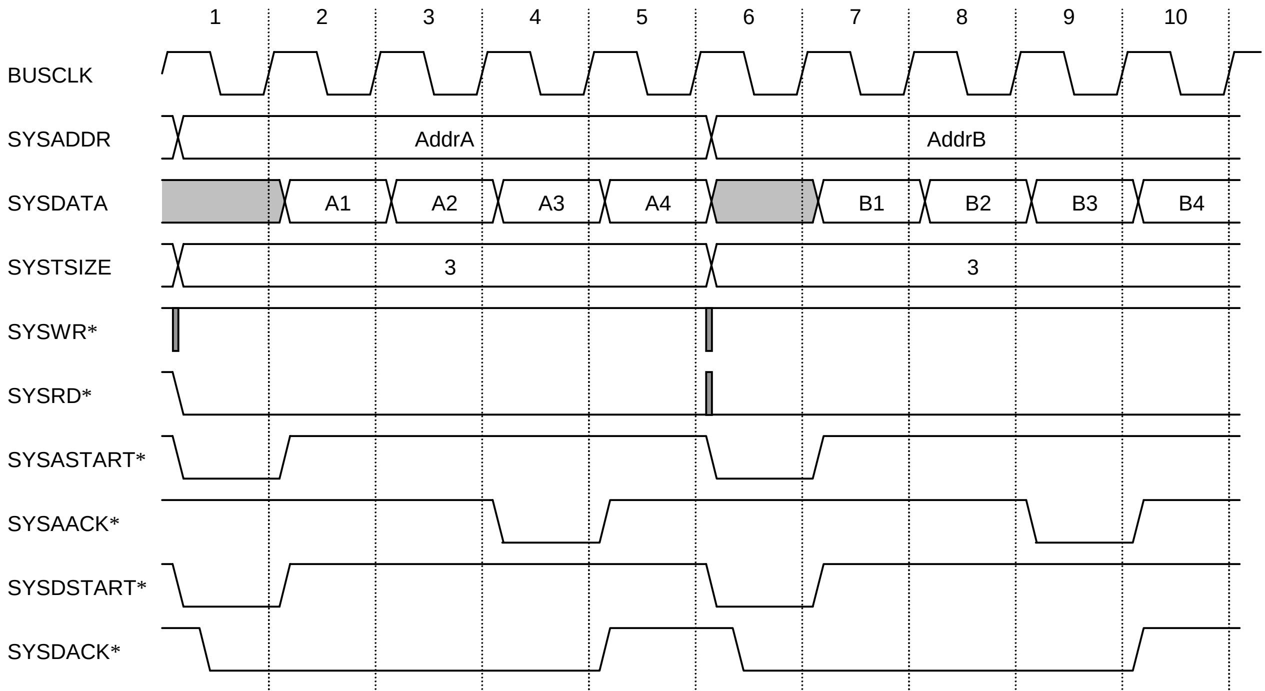


Figure 8-16. CPU Non-Pipeline Burst Reads

8.5.5.2 CPU Non-Pipeline Burst Writes

All adjacent address phases are write cycles.

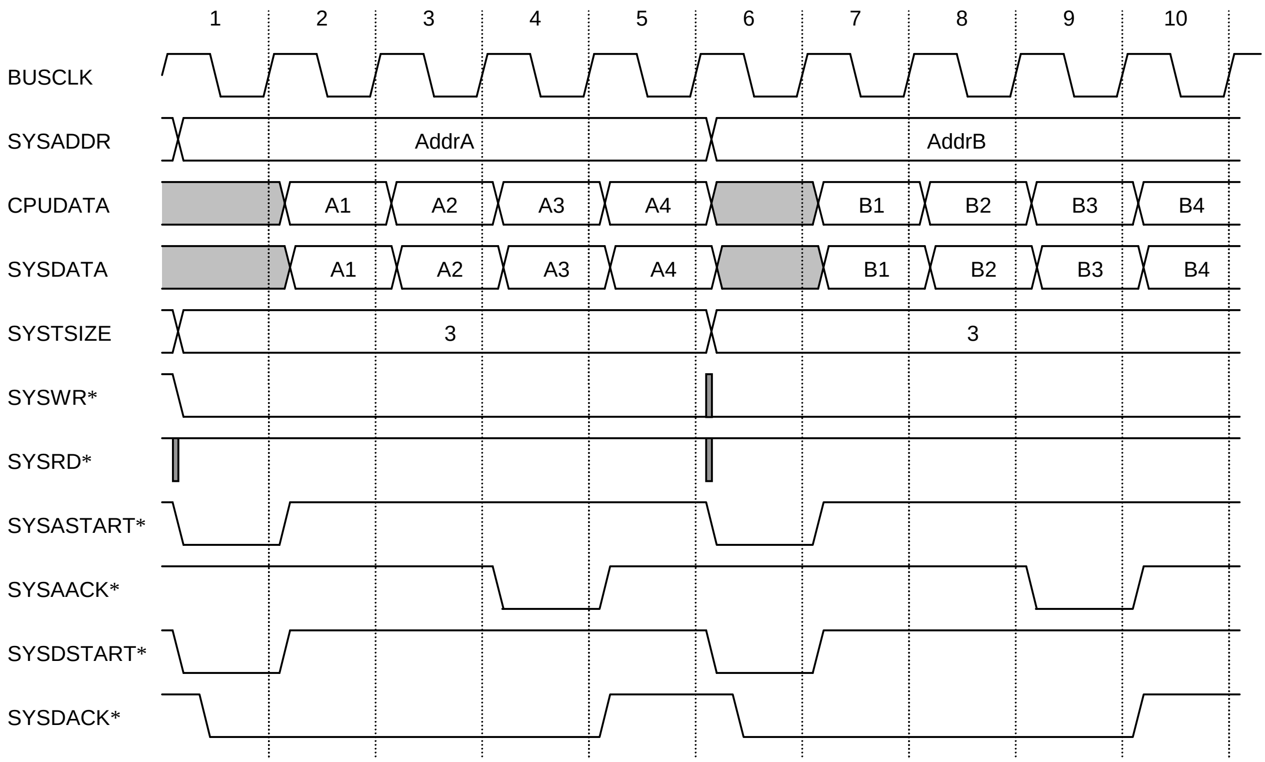


Figure 8-17. CPU Non-Pipeline Burst Writes

8.5.6 Bus Error Operations

Bus error occurs when there are no slave responding to the address or data phases of the bus cycle. When bus error occurs, the current bus operation is terminated, and the system proceeds with the next bus operation. Without bus error detection, the CPU Bus would remain waiting indefinitely for the SYSAACK* or SYSDACK* signals.

Bus error is generated by the CPU Bus monitor logic. The monitor logic basically makes sure that for both address and data phases in the current CPU Bus cycle, there are SYSAACK* and SYSDACK*, respectively. In the case, when there is no SYSAACK* or SYSDACK* or response to the address or data phase for a pre-defined period of time for the current CPU Bus cycle, bus error is generated by asserting BUSERR* for one CPU Bus clock. Bus error has higher priority than SYSAACK* or SYSDACK* if they are detected in the same cycle.

Bus error is always asserted in reference to the data phase of the cycle. The exact timing is the cycles from SYSDSTART* asserted to the cycle before the assertion of the next SYSDSTART*. The bus error signal is sampled when the system is waiting for the assertion of SYSDACK* and/or SYSAACK* of the operation corresponding to the current data phase. For example, if the address phase of a certain cycle has no response from the slave devices, the bus monitor logic will wait until the SYSDSTART* of the corresponding data phase before generating the bus error. The bus monitor logic can generate the bus error any time before the next data phase begins.

8.5.6.1 Bus Error Exceptions

As mentioned before, two operations can be pipelined on the CPU bus, and these two operations can be initiated from either the CPU as master or the DMAC as master.

If the bus error occurs in the CPU initiated operation, the following occurs:

- a bus error exception due to instruction fetch or data access is generated
- the bus error instruction or data address is recorded in the *BadPAddr* Register of COP0
- the *Status.BEM* bit is set (This bit is the bus error mask (BEM) in the COP0 Status Register).

Once a bus error occurs, any further bus errors are ignored until *Status.BEM* is cleared by the bus error exception handler.

If the bus error occurs in the DMA initiated operation (DMA cycle), the DMAC will finish the pending pipeline operations, disable itself, release the CPU Bus, and cause an interrupt. The interrupt routine will then service and re-enable the DMAC accordingly. Table 8-4 summarizes the exception generation:

Table 8-4. Bus Error Exceptions

Operation with the Bus Error	Exception Generated
CPU Initiated Instruction Fetch	Bus Error Exception - Instruction Fetch
CPU Initiated Data Access	Bus Error Exception - Data Access
DMA Cycle	Interrupt Exception

8.5.6.2 CPU Bus Cycle Termination

Two pipeline operations can be in progress at any time, but if a bus error occurs, only the operation with the bus error is terminated. That is, the occurrence of a bus error with one master does not affect the program execution of another master. For example, if bus error occurs when the first and second operations are initiated from the DMAC and CPU, respectively, the CPU Bus will terminate the DMA operation and continue with the CPU operation. Table 8-5 summarizes CPU Bus cycle sequence for all types of CPU Bus cycle termination.

Table 8-5. Operation Termination Sequence

First Operation with Bus Error	Second Operation	CPU Bus Cycle Sequence
CPU Cycle #1	CPU Cycle #2	1. CPU Cycle #1 is terminated. 2. Bus Error Exception occurs. 3. CPU Cycle #2 continues on.
CPU Cycle #1	DMA Cycle #2	1. CPU Cycle #1 is terminated. 2. Bus Error Exception occurs. 3. DMA Cycle #2 continues on.
DMA Cycle #1	CPU Cycle #2	1. DMA Cycle #1 is terminated. 2. CPU Cycle #2 continues on. 3. DMA releases CPU Bus, disable itself (disable further requests until the interrupt routine re-enable the DMAC), and generate an interrupt. 4. CPU cycles continues on.
DMA Cycle #1	DMA Cycle #2	1. DMA Cycle #1 is terminated. 2. DMA Cycle #2 continues on. 3. DMAC releases CPU Bus, disable itself (disable further requests until the interrupt routine re-enable the DMAC), and generate an interrupt. 4. CPU cycles continue on.

8.5.6.3 Bus Error Timing with No Pending Operation

If there are no pending operations on the bus, BUSERR* is ignored at all times.

8.5.6.4 Bus Error Timing with One Pending Operation

If there is one pending operation on the bus, BUSERR* is sampled while waiting for the assertion of SYSAACK* or SYSDACK*. If BUSERR* is asserted, the bus cycle will continue as if the SYSAACK* and/or the last SYSDACK* has been asserted. Figure 8-18, Figure 8-19, and Figure 8-20 illustrates the bus error associated with one pending operation. In these figures, BUSERR* is ignored before CPUDSTART* and after BUSERR* asserted because the bus is not waiting for the assertion of SYSAACK* nor SYSDACK*.

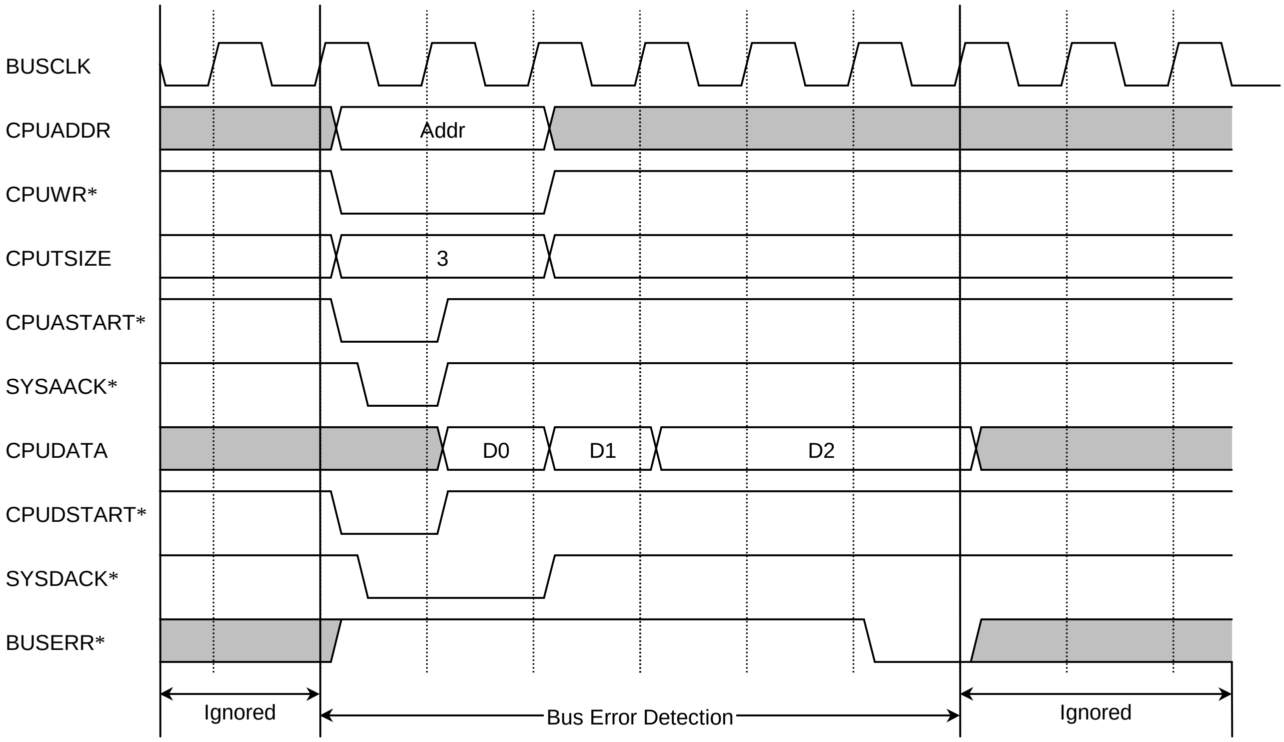


Figure 8-18. One Operation with BUSERR* as the Last SYSDACK*

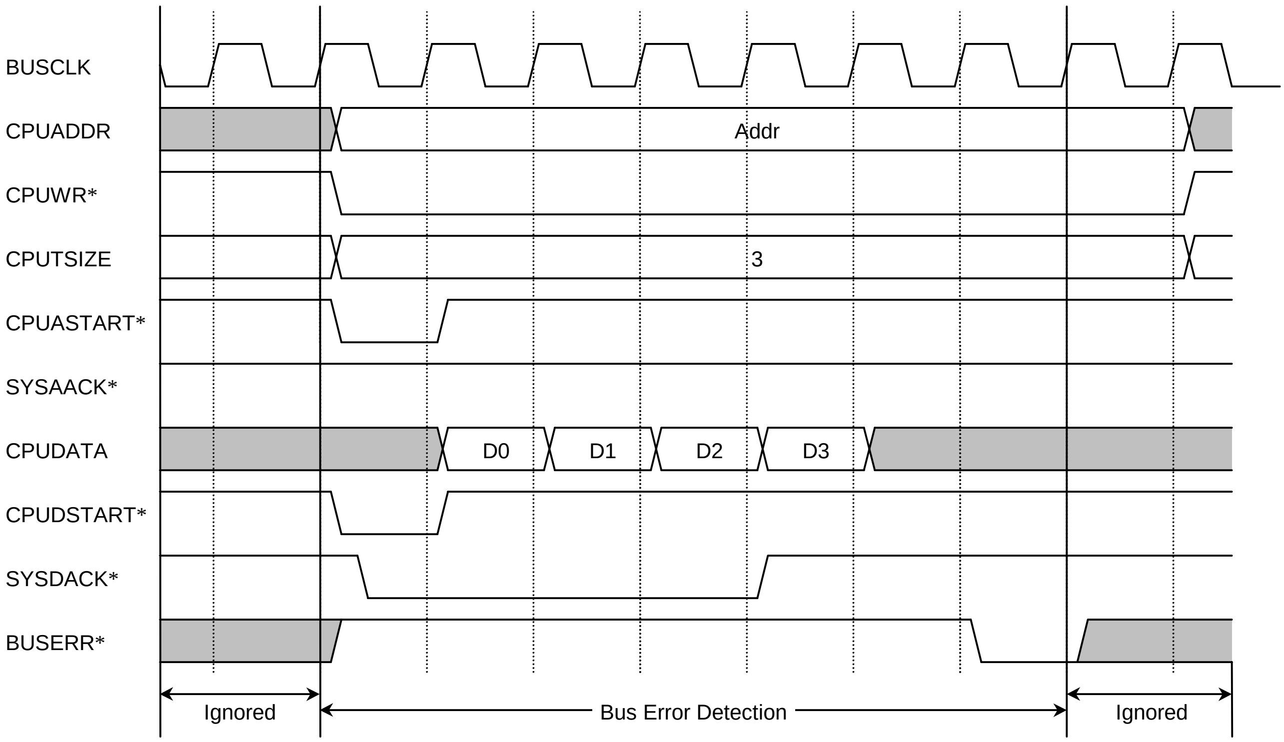


Figure 8-19. One Operation with BUSERR* as SYSAACK*

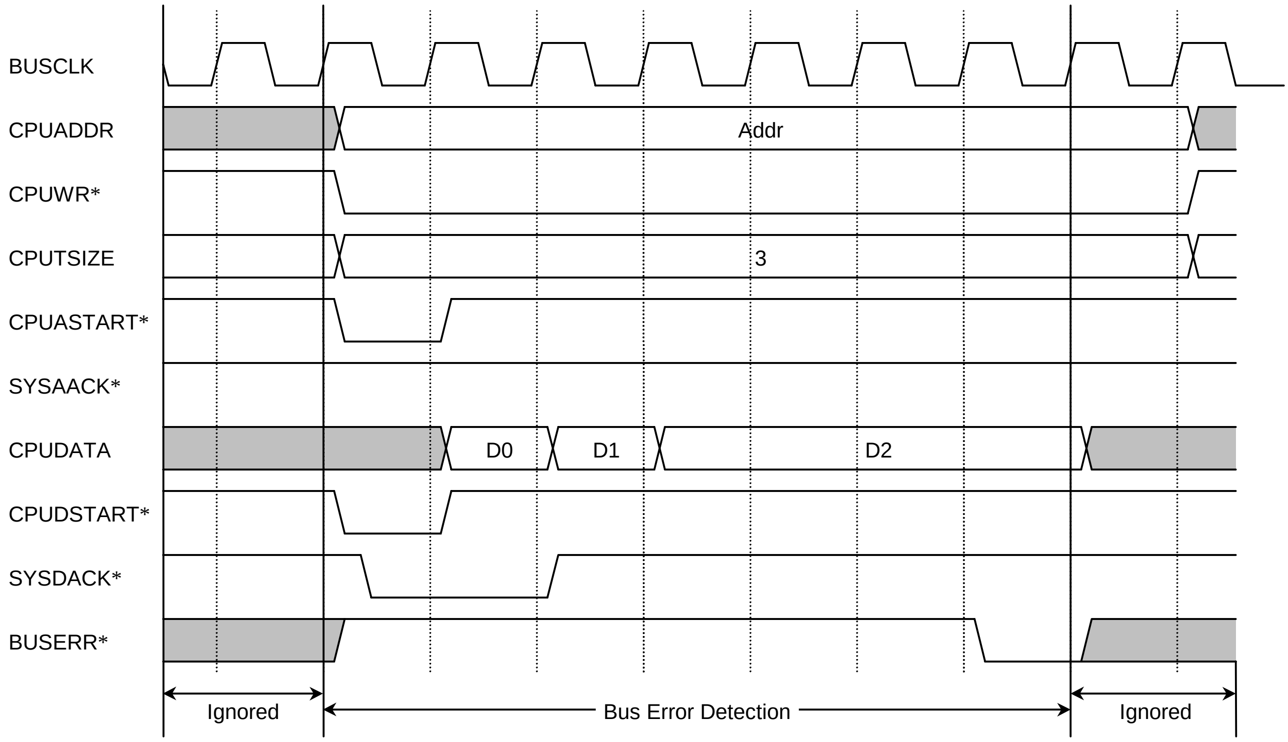


Figure 8-20. One Operation with BUSERR* as SYSAACK*
and the Last SYSDACK*

8.5.6.5 Bus Error Timing with Two Pending Operations

If there are two pending operations on the bus, BUSERR* is sampled while waiting for the assertion of SYSDACK*. If BUSERR* is asserted, the bus cycle will continue as if the last SYSDACK* has been asserted. The bus cycle will then proceed with the data phase of the next operation. The bus error that occurred is for the first pending operation.

Figure 8-21 illustrates the bus error associated with two pending operations. In this figure, BUSERR* is ignored after BUSERR* asserted because the bus is no longer waiting for the assertion of SYSDACK* corresponding to operation AddrA with the bus error, and detection of bus error for operation AddrB has not started until the assertion of CPUDSTART*.

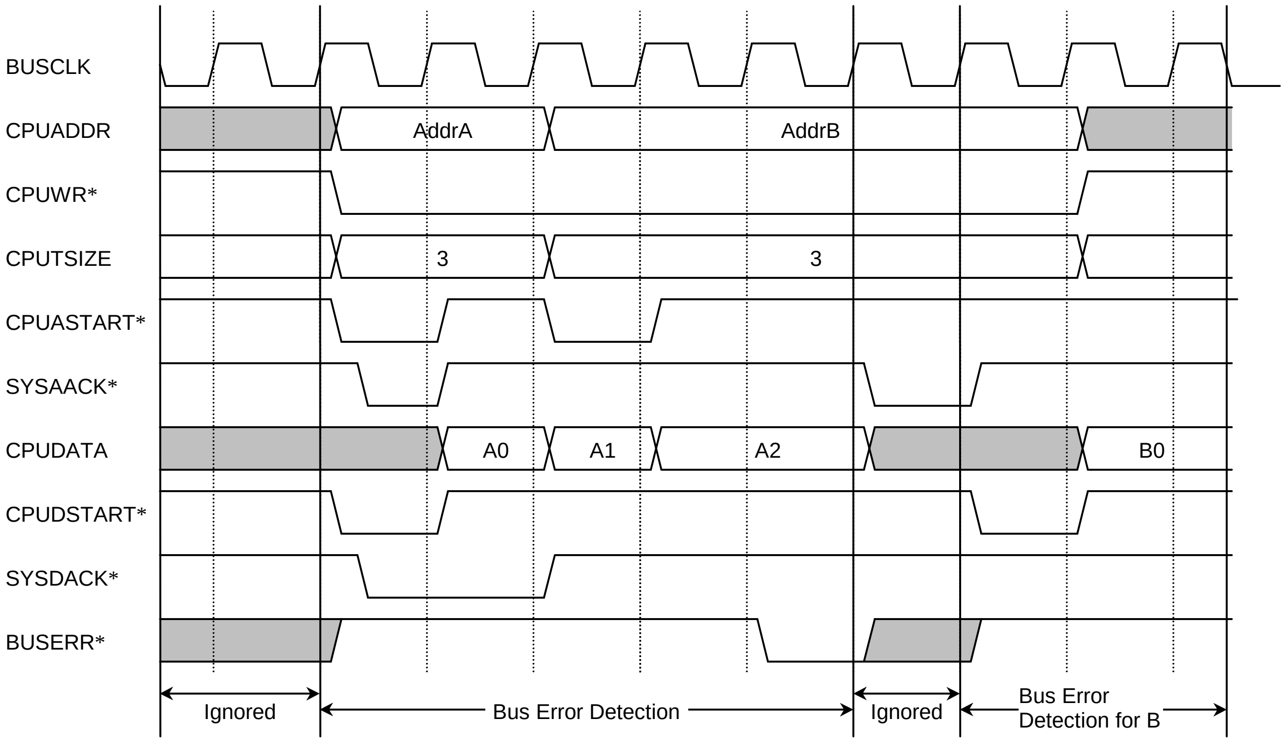


Figure 8-21. Two Operations with Bus Error as the Last SYSDACK*

9. Performance Counter

The performance counter provides the means for gathering statistical information about the internal events of the CPU and the pipeline during program execution. The statistics gathered during program execution aid in tuning the performance of hardware and software systems based on the processor.

9.1 Overview

The performance counter consists of one control register and two counters. The control register controls the functions of the monitor while the counters count the number of events specified by the control register.

9.2 Performance Counters and Performance Control Registers

The *Performance Counter Control Register*, or *PCCR*, and *Performance Counter Registers PCR0* and *PCR1* are mapped into *COP0* Register 25. Both the register and counters are read/write registers accessible by *MTPC*, *MTPS*, *MTC0*, *MFPC*, *MFPS* and *MFC0* instructions. Each counter is capable of counting one event as specified by the control register.

The format of the *PCCR* is shown in Figure 9-1, and the format of *PCR0* and *PCR1* is shown in Figure 9-2.

31	30	29	28	27	26	25	24	23	22	21	20	19 15	14	13	12	11	10	9 5	4	3	2	1	0
CTE	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**EVENT1**	**U1**	**S1**	**K1**	**EXL1**	**0**	**EVENT0**	**U0**	**S0**	**K0**	**EXL0**	**0**
1	1	1	1	1	1	1	1	1	1	1	1	5	1	1	1	1	1	5	1	1	1	1	1

Figure 9-1. Format of the Performance Counter Control Register PCCR

31	30 0
OVFL	**VALUE**
1	31

Figure 9-2. Format of Performance Counter Registers PCR0 and PCR1

The interpretation of the *PCCR* register bits is as follows:

Table 9-1. PCCR Register Bits

Field	Function	Initial Value
CTE	If 1, PCR0 and PCR1 counting and exception generation is enabled.	0
EVENT0/1	Event counted by PCR0/1; see Table 9-5 for details.	Undefined
U0/1	PCR0/1 counts event EVENT0/1 when in User mode.	Undefined
S0/1	PCR0/1 counts event EVENT0/1 when in Supervisor mode.	Undefined
K0/1	PCR0/1 counts event EVENT0/1 when in non-exception Kernel mode; i.e. with both STATUS.EXL and STATUS.ERL set to 0.	Undefined
EXL0/1	PCR0/1 counts event EVENT0/1 when in Level 1 exception handler.	Undefined

9.2.1 Accessing Counters and Registers

The counter control register *PCCR* and the two performance counter registers *PCR0* and *PCR1* are accessed by using *MTC0** and *MFC0** instructions. All three registers are mapped to *COP0* register 25. Table 9-2 illustrates how these registers are written by using the *MTC0* instruction, and Table 9-3 illustrates the encoding of the *MFC0* instructions used to read the registers.
Table 9-4 show special mnemonics to access the performance Counters and Registers.

Table 9-2. Writing Performance Counters and Registers using MTC0

OpCode[15:11]	OpCode[1:0]	Operation
11001	00	Move to Counter Control Register
11001	01	Move to Performance Counter Register 0
11001	10	*unused*
11001	11	Move to Performance Counter Register 1

Table 9-3. Reading Performance Counters and Registers using MFC0

OpCode[15:11]	OpCode[1:0]	Operation
11001	00	Move from Counter Control Register
11001	01	Move from Performance Counter Register 0
11001	10	*unused*
11001	11	Move from Performance Counter Register 1

Table 9-4. Mnemonics to Access the Performance Counters and Registers

MTPC	Move to Performance Counter
MTPS	Move to Performance Event Specifies
MFPC	Move from Performance Counter
MFPS	Move from Performance Event Specifies

[*] MTPC, MTPS, MFPC and MFPS are the special encoding of MTC0 and MFC0.

9.2.2 State of Performance Counter Control Registers Upon Reset

The CTE bit of the *Performance Counter Control Register PCCR* is initialized to 0 upon reset. This prevents event counting and interrupt generation until the control registers are initialized. It also allows a precise way for counters to be initialized by software; see the section 9.3.2 for more details. Note that the remaining bits of *PCCR* and both registers *PCR0* and *PCR1* must be initialized by software.

9.3 Counter Operation

The performance counters *PCR0* and *PCR1* increment by 1 whenever their corresponding count event occurs, and the counter is enabled. The count event for *PCR0* is specified by *PCCR.EVENT0* and the count event for *PCR1* is specified by *PCCR.EVENT1*. The encoding of the *EVENT* field is specified in Table 9-5, and discussed in detail later. A counter is enabled only when both of the following conditions are satisfied:

1. The global counter enable flag *PCCR.CTE* is set to 1, and

2. The current privilege mode matches the permitted privilege mode for each counter. The values in *PCCR.U0*, *PCCR.S0*, *PCCR.K0*, and *PCCR.EXL0* specify the permitted privilege modes for *PCR0* and *PCCR.U1*.
PCCR.S1, *PCCR.K1*, and *PCCR.EXL1* specify the permitted privilege modes for *PCR1*. For example, if the current privilege mode is *SUPERVISOR*, *PCR0* will operate only if *PCCR.S0* is set to 1. Note that there is no “ERL0” or “ERL1” flag in *PCCR*. This is because counters are unconditionally disabled when in level 2 handlers.

9.3.1 Counter Events

A counter increments if it is enabled and its trigger event occurs. The permissible values for *PCCR.EVENT0* and *PCCR.EVENT1* are as shown in Table 9-5 below. The events are described in Section.9.3.1.1Event Descriptions

Table 9-5. Counter Events

Event	Counter 0	Counter 1
0	reserved	Low-order branch issued
1	Processor cycle	Processor cycle
2	Single instruction issue	Dual instruction issue
3	Branch issued	Branch mispredicted
4	BTAC miss	JTLB miss
5	ITLB miss	DTLB miss
6	I$ miss	D$ miss
7	DTLB accessed	WBB single request unavailable
8	Non-blocking load/store	WBB burst request unavailable
9	WBB single request	WBB burst request almost full
10	WBB burst request	WBB burst request full
11	CPU address bus busy	CPU data bus busy
12	Instruction completed	Instruction completed
13	Non-BDS instruction completed	Non-BDS instruction completed
14	reserved	COP1 instruction completed
15	Load completed	Store completed
16	No event	No event
17-31	reserved	reserved

9.3.1.1 Event Descriptions

In event descriptions, the word 'branch' (for example, 'branch issued', or 'branch miss-predicted') means any 'transfer of control' instruction that is subject to prediction (that is, all the conditional branch instructions, *J*, and *JAL*). The *JR*, *JALR*, *ERET*, *SYSCALL*, *BREAK*, and *TRAP* instructions are not included.

Branch issued	This event is triggered whenever a branch is issued to a functional pipe. Note that a branch that is issued in a pipelined implementation may get canceled if an instruction prior to it signals an exception.
Branch mispredicted	This event is triggered whenever the predicted branch address (taken or not-taken) is incorrect. Note that a branch that is issued in a pipelined implementation may get canceled if an instruction prior to it signals an exception.
BTAC miss	This event is triggered whenever the instruction address lookup into the BTAC fails. Counts low-order (even) branch instructions that miss the BTAC. Note that high-order (odd) branch does not refer the BTAC.
COP1 instruction completed	This event is triggered when a COP1 instruction completes. The event is signaled even if the COP1 instruction completes successfully, but appears in the branch delay slot of a branch-likely instruction and is therefore nullified.
CPU address bus busy	Generates a signal once every BUSCLK (not CPU clock) that the CPU address bus is unavailable. The CPU address bus is considered unavailable whenever it is busy, or when two addresses have been issued but the data for the first address has yet to return.
Data cache miss	This event is triggered whenever a data cache miss is detected. See Table 9-6. for the D$ miss definition.

Table 9-6. Definition of Data Cache Miss

Access	DCE	Page Attr.	Hit/Miss
Load	0	Uncached, UCA, Cached	Miss
	1	Uncached, UCA	Miss
		Cached	Hit/Miss
Store	0	Uncached, UCA, Cached	Hit
	1	Uncached, UCA	Hit
		Cached	Hit/Miss
Pref	0	Uncached, UCA, Cached	Uncount *
	1	Uncached, UCA	Uncount *
		Cached	Hit/Miss

In this event, the data cache miss is defined as any load/store/pref instructions which may generate bus read operations to get missed data from external memory.

* Prefetch to the Uncached or UCA page is considered as nop.

DTLB accessed	Barring canceled instructions, this event counts the total number of executed loads and stores. Thus, 'data cache miss' divided by 'DTLB accessed' provide a good estimate of the D miss rate (assuming no uncached loads/stores occur). Also, 'DTLB miss' divided by 'DTLB accessed' provides the DTLB miss rate. DTLB is accessed even when unmapped page is accessed in case that minor revision number is 0x10 or later.
DTLB Miss	This event is triggered whenever a DTLB miss is detected. DTLB is accessed even when unmapped page is accessed in case that minor revision number is 0x10 or later.
Dual instruction issued	This event is signaled whenever both functional pipes of the C790 are issued instructions*. The event counter is incremented by 1.
Instruction cache miss	This event is triggered whenever an instruction cache miss is detected.
Instruction completed	This event triggers when an instruction completes. Note that some instructions (e.g. SYSCALL, TEQ, TEQI, etc.) signal exceptions as a normal part of their operation. Such instructions are considered complete whether or not the "normal" exception was raised. Therefore, an "instruction complete" event is signaled even if a TEQ succeeds (i.e. raises a Trap exception). However, if a "true" exception occurs (e.g. a counter exception is signaled while the TEQ is executing), the instruction is canceled and no "instruction complete" signal is generated. Similarly, an instruction in the branch delay slot (BDS) of a branch-likely instruction is counted as complete even if the BDS instruction is nullified. If the BDS instruction is canceled because of a "true" exception, no "instruction completed" event is signaled. C790 Implementation Note: Up to two instructions can complete every cycle in the C790. When two instructions do complete, the event counter is incremented by 2.
ITLB miss	This event is triggered whenever a ITLB miss is detected.
JTLB miss	This event is triggered whenever a JTLB miss is detected.
Load completed	This event triggers when a load instruction completes. Note that the event is signaled even if the load appears in the branch delay slot of a branch-likely instruction that is not taken and is therefore nullified.
Low-order branch issued	Counts the numbers of branches that were issued that appeared in the low-order (even) position of an instruction pair fetch. This count is needed since only these branches are subject to BTAC lookup.
No event	This "event" effectively disables the corresponding counter. It is useful principally if only one of the two counters need be activated.
Non-BDS instruction completed (for stepping)	This event triggers when an instruction that does not have a branch delay slot completes. In particular, it does not trigger when a branch or jump instruction completes. However, it does trigger when the instruction in the branch delay slot of the branch or jump completes. In the case of a branch-likely instruction, the instruction in the branch delay slot triggers the event even if this instruction is nullified. Note: this event is useful for stepping over instructions.

* (Dual instruction issued) *2 + (Single instruction issued) = instruction issued
(Instruction issued) – (instruction completed) = instruction canceled

Non-blocking load/store (1st cache miss):	This event is signaled whenever a cached load/store/pref instruction misses on the Data Cache and there is no pending data cache miss, UCAB miss and uncached load.
Processor cycle	This event triggers on every processor clock cycle.
Single instruction issued	This event is signaled whenever only one of the functional pipes of the C790 is issued an instruction*.
Store completed	This event triggers when a store instruction completes. Note that the event is signaled even if the store appears in the branch delay slot of a branch-likely instruction that is not taken and is therefore nullified.
WBB Single Request	A non-burst request was made to the WBB.
WBB Burst Request	A burst request was made to the WBB.
WBB Single Request unavailable	A non-burst request was made to the WBB, but there were insufficient free entries in the WBB to service it. All 8 entries are used at that time.
WBB Burst Request unavailable	A burst request was made to the WBB, but, the WBB was completely full, or there were not enough to service the request. 5, 6, 7, 8 entries are used at that time.
WBB Burst Request almost full	A burst request was made to the WBB, and even though there were free entries, there were not enough to service the request. 5, 6, 7 entries are used at that time.
WBB Burst Request full	A burst request was made to the WBB, but the WBB was completely full. All 8 entries are used at that time.

* (Dual instruction issued) *2 + (Single instruction issued) = instruction issued
(Instruction issued) – (instruction completed) = instruction canceled

9.3.2 Handling Performance Counter Exceptions

A performance counter exception is detected by an instruction if the following condition holds true:

```
~STATUS.ERL && PCCR.CTE && (CTR0.OVFL || CTR1.OVFL)
```

Note that software should not rely on the exception occurring if the instruction is nullified; i.e. it appears in the branch delay slot of a branch likely instruction that is not taken.

C790 Implementation Note: C790 implementation always counts events that occur within nullified instructions.

The instruction detecting a counter exception is canceled by the exception, and instruction execution continues as follows:

```
if ( in branch delay slot ) {
  ErrorEPC = PC - 4;
  CAUSE.BD2 = 1;
}
else {
  ErrorEPC = PC;
  CAUSE.BD2 = 0;
}
if ( STATUS.DEV )
  PC = 0xBFC00280;  // Uncached counter xcp handler
else
  PC = 0x80000080;  // “Normal” counter xcp handler
STATUS.ERL = 1;
CAUSE.EXC2 = 2;     // Counter exception
```

The description above makes use of the *BD2* and *EXC2* fields in the *CAUSE* register. Both are fields newly introduced in the C790 and occupy the bit positions shown below.

31	30	29-28	27	26	25	24	23	22	21	20	19	18-16	15	14	13	12	11	10	9	8	7	6-2	1	0
BD	**BD2**	CE	0	0	0	0	0	0	0	0	0	**EXC2**	IP7	0	0	SIOP	IP3	IP2	0	0	0	EXC	0	0

Figure 9-3. CAUSE Register Fields

C790 Programming Note: Note that the “normal” exception entry point is in kseg0 space. That is, the address is unmapped and the caching policy is determined by *CONFIG.K0*. If you don’t want to disturb the cache while counting and stepping, kseg0 should be configured in “uncached” mode. If cache data preservation is secondary to counter exception servicing performance counter overflow, kseg0 should be configured in “cached” mode.

9.3.3 Priority of Counter Exceptions

Counter exceptions have the highest priority after cold reset and NMI. If a cold reset occurs the processor is initialized – so a simultaneous counter exception is discarded. If an NMI occurs, the NMI handler is entered with either *PCR0.OVFL* or *PCR1.OVFL* (or both) set to 1, and *ErrorEPC* pointing at the instruction causing the counter overflow. (*ErrorEPC* is used because NMI is handled as a level 2 exception.) Once the NMI handler exits, the instruction that caused the overflow is re-executed. However, since *PCR0.OVFL* or *PCR1.OVFL* is 1, the instruction is canceled once more and the counter exception handler is entered.

9.3.4 Initializing Counters

Let us look at the code sequence needed to initialize counters and activate them. In the example below, *PCR0* is set up to count clocks in all operating modes and report a counter exception after the count exceeds 2^{31}. *CTR1* is set up to count stores while in supervisor mode only, and report a counter exception after the count exceeds 2^{31}. The code must be executed while in level 2 exception mode (ERL=1).

```
STATUS.ERL = 1;          // Set ERL (to inhibit counting)
ErrorEPC = <target instruction where counting is to start>

PCR0 = 0;            // Init CTR0, and …
PCCR.EVENT0 = 1;         // … set up to count clocks …
PCCR.U0 = 1;         // … in all privilege modes
PCCR.S0 = 1;
PCCR.K0 = 1;
PCCR.EXL0 = 1;

PCR1 = 0;            // Init PCRT1, and …
PCCR.EVENT1 = 15;        // … set up to count completed stores …
PCCR.U1 = 0;         // … while in supervisor mode
PCCR.S1 = 1;
PCCR.K1 = 0;
PCCR.EXL1 = 0;

PCCR.CTE = 1;  // Enable global counter flag
ERET          // Execute ERET to clear ERL -
              // counting begins with ERET's target
              // Note that the ERET instruction also
              // guarantees that the COP0 state
              // updated (e.g. CCR) is valid.
```

9.3.5 The Note to Read Counters

Whenever you want to read a counter by MTC0 or MTPC, be sure that any counting events must NOT occur, otherwise you may get wrong number. For example, counter for TLB event should be read in the unmapped area, that of instruction completion event should be read in the ERL=1 (level 2 exception) area or other disabled area.

It is a implement-dependent that when the event is counted. It depends on the number of the pipeline stages and so on.

To write a robust code among silicon versions and mask versions, you read the counters after flushing the pipeline by *SYNC.P* instruction. C790 is a pipeline processor. It is required for the instruction completion type event.

It is a nature of event counting that some inaccuracy exists. You don't need to be surprised if different number is observed in different version of silicon/mask.

10. Floating-Point Unit, CP1 (Option)

This chapter describes the floating-point operations, including the programming model, instruction set and formats.

The floating-point operations fully conform to the requirements of ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*.

10.1 Overview

All floating-point instructions, as defined in the MIPS ISA for the floating-point coprocessor, CP1, are processed by the other hardware unit that executes integer instructions.

The floating point execution unit can be disabled by the coprocessor usability *CU* bit defined in the CP0 *Status* register.

10.2 Floating Point Register

10.2.1 Floating-Point General Registers (FGRs)

CP1 has a set of *Floating-Point General Purpose registers (FGRs)* that can be accessed in the following ways:

- As 32 general purpose registers (32 FGRs), each of which is 32 bits wide when the *FR* bit in the *CPU* Status register equals 0; or as 32 general purpose registers (32 FGRs), each of which is 64-bits wide when FR equals 1. The CPU accesses these registers through move, load, and store instructions.
- As 16 floating-point registers (see the next section for a description of FPRs), each of which is 64-bits wide, when the *FR* bit in the CPU *Status* register equals 0. The FPRs hold values in either single- or double-precision floating-point format. Each FPR corresponds to adjacently numbered FGRs as shown in Figure 10-1.
- As 32 floating-point registers (see the next section for a description of FPRs), each of which is 64-bits wide, when the *FR* bit in the CPU *Status* register equals 1. The FPRs hold values in either single- or double-precision floating-point format. Each FPR corresponds to an FGR as shown in Figure 10-1.

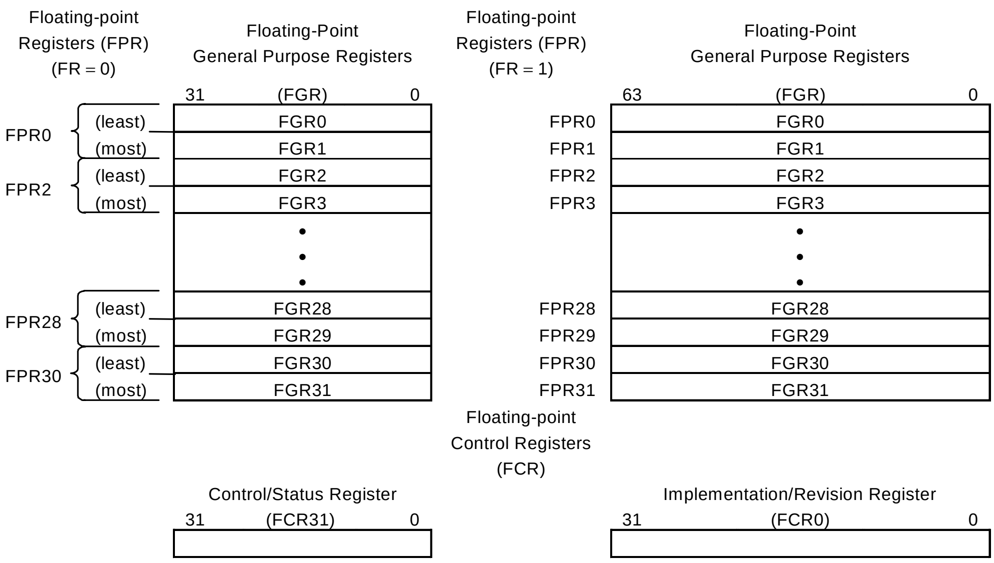


Figure 10-1. FP Registers

10.2.2 Floating-Point Registers (FPRs)

The FPU provides:

- 16 *Floating-Point* registers (*FPRs*) when the *FR* bit in the *Status* register equals 0, or
- 32 *Floating-Point* registers (*FPRs*) when the *FR* bit in the *Status* register equals 1.

These 64-bit registers hold floating-point values during floating-point operations and are physically formed from the *General Purpose* registers (*FGRs*). When the *FR* bit in the *Status* register equals 1, the *FPR* references a single 64-bit *FGR*.

The *FPRs* hold values in either single- or double-precision floating-point format. If the *FR* bit equals 0, only even numbers (the *least* register) can be used to address *FPRs*. When the *FR* bit is set to a 1, all *FPR* register numbers are valid.

If the *FR* bit equals 0 during a double-precision floating-point operation, the general registers are accessed in double pairs. Thus, in a double-precision operation, selecting *Floating-Point Register 0 (FPR0)* actually addresses adjacent *Floating-Point General Purpose* registers *FGR0* and *FGR1*.

10.2.3 Floating-Point Control Registers

The MIPS RISC architecture defines 32 floating-point control registers (*FCRs*); the C790 processor implements two of these registers: *FCR0* and *FCR31*. These *FCRs* are described below:

- The *Implementation/Revision* register *(FCR0)* holds revision information.
- The *Control/Status* register *(FCR31)* controls and monitors exceptions, holds the result of compare operations, and establishes rounding modes.
- *FCR1* to *FCR30* are reserved.

Table 10-1 lists the assignments of the FCRs.

Table 10-1. Floating-Point Control Register Assignments

FCR Number	Use
FCR0	Coprocessor implementation and revision register
FCR1 to FCR30	Reserved
FCR31	Rounding mode, cause, trap enables, and flags

Implementation and Revision Register (FCR0)

The read-only *Implementation and Revision* register *(FCR0)* specifies the implementation and revision number of CP1. This information can determine the coprocessor revision and performance level, and can also be used by diagnostic software.

Figure 10-2 shows the layout of the register; Table 10-2 describes the *Implementation and Revision* register *(FCR0)* fields.

Implementation/Revision Register (FCR0)

31 ... 16	15 ... 8	7 ... 0
0	Imp	Rev
16	8	8

Figure 10-2. Implementation/Revision Register

Table 10-2. FCR0 Fields

Field	Description	Initial value
Imp	Implementation number	0x38
Rev	Revision number in the form of y. x	Revision Number
0	Reserved. Returns zeroes when read.	

The revision number is a value of the form *y. x*, where:

- *y* is a major revision number held in bits 7:4.
- *x* is a minor revision number held in bits 3:0.

The revision number distinguishes some chip revisions; however, there is not guarantee that changes to its chips are necessarily reflected by the revision number, or that changes to the revision number necessarily reflect real chip changes. For this reason revision number values are not listed, and software should not rely on the revision number to characterize the chip.

IEEE Standard 754

IEEE Standard 754 specifies that floating-point operations detect certain exceptional cases, raise flags, and can invoke an exception handler when an exception occurs. These features are implemented in the MIPS architecture with the *Cause*, *Enable*, and *Flag* fields of the *Control/Status* register. The *Flag* bits implement IEEE 754 exception status flags, and the *Cause* and *Enable* bits implement exception handling.

Control/Status Register (FCR31)

The *Control/Status* register *(FCR31)* contains control and status information that can be accessed by instructions in either Kernel or User mode. *FCR31* also controls the arithmetic rounding mode and enables User mode traps, as well as identifying any exceptions that may have occurred in the most recently executed floating-point instruction, along with any exceptions that may have occurred without being trapped.

Figure 10-3 shows the format of the *Control/Status* register, and Table 10-3 describes the *Control/Status* register fields. Figure 10-4 shows the *Control/Status* register *Cause, Flag,* and *Enable* fields.

Control/Status Register (FCR31)

31 25	24	23	22 18	17 12	11 7	6 2	1 0
0	FS	C	0	Cause EVZOUI	Enables VZOUI	Flags VZOUI	RM
7	1	1	5	6	5	5	2

Figure 10-3. FP Control/Status Register Bit Assignments

Table 10-3. Control/Status Register Fields

Field	Description
FS	When set, denormalized results can be flushed instead of causing an unimplemented operation exception.
C	Condition bit. See description of *Control/Status* register *Condition* bit.
Cause	Cause bits. See Figure 10-4 and the description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
Enables	Enable bits. See Figure 10-4 and the description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
Flags	Flag bits. See Figure 10-4 and the description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
RM	Rounding mode bits. See Table 10-5 and the description of *Control/Status* register *Rounding Mode Control* bits.

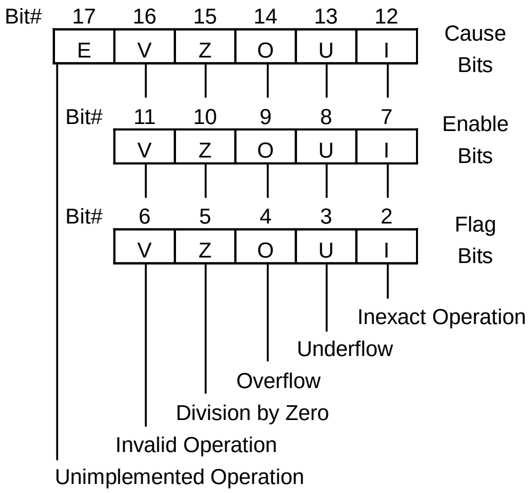


Figure 10-4. Control/Status Register Cause, Flag, and Enable Fields

Control/Status Register FS Bit

The *FS* bit enables the flushing of denormalized values. When the *FS* bit is set and the Underflow and Inexact *Enable* bits are not set, denormalized results are flushed instead of causing an Unimplemented Operation exception. Results are flushed to either 0 or the minimum normalized value, depending upon the rounding mode (see Table 10-4 below), and the Underflow and Inexact of the *Cause* and *Flag* bits are set.

Table 10-4. Flush Values of Denormalized Results

Denormalized Result	Flushed Result Rounding Mode			
	RN	RZ	RP	RM
Positive	+0	+0	$+2^{Emin}$	+0
Negative	-0	-0	-0	-2^{Emin}

Control/Status Register Condition Bit

When a floating-point Compare operation takes place, the result is stored at bit 23, the *Condition* bit. The *C* bit is set to 1 if the condition is true; the bit is cleared to 0 if the condition is false. Bit 23 is affected only by compare and *CTC1* instructions.

Control/Status Register Cause, Flag, and Enable Fields

Figure 10-4 illustrates the *Cause, Flag,* and *Enable* fields of the *Control/Status* register. The *Cause* and *Flag* fields are updated by all conversion, computational (except MOV. fmt), *CTC1*, reserved, and unimplemented instructions. All other instructions have no affect on these fields.

Cause Bits

Bits 17:12 in the *Control/Status* register contain *Cause* bits, as shown in Figure 10-4, which reflect the results of the most recently executed floating-point instruction. The *Cause* bits are a logical extension of the CP0 *Cause* register; they identify the exceptions raised by the last floating-point operation. If the corresponding *Enable* bit is set at the time of the exception a floating-point exception is raised and trapped by CPU. If more than one exception occurs on a single instruction, each appropriate bit is set.

The *Cause* bits are updated by most floating-point operations. The Unimplemented Operation *(E)* bit is set to 1 if software emulation is required, otherwise it remains 0. The other bits are set to 0 or 1 to indicate the occurrence or non-occurrence (respectively) of an IEEE 754 exception. Within the set of floating-point instructions that update the *Cause* bits, the *Cause* field indicates the exceptions raised by the most-recently-executed instruction.

When a floating-point exception is taken, no results are stored, and the only state affected is the *Cause* bit.

Enable Bits

A floating-point exception is generated any time a *Cause* bit and the corresponding *Enable* bit are set. A floating-point operation that sets an enabled *Cause* bit forces an immediate floating-point exception, as does setting both *Cause* and *Enable* bits with *CTC1*.

There is no enable for Unimplemented Operation *(E)*. An Unimplemented exception always generates a floating-point exception.

Before returning from a floating-point exception, software must first clear the enabled *Cause* bits with a *CTC1* instruction to prevent a repeat of the exception trapping. Thus, User mode programs can never observe enabled *Cause* bits set; if this information is required in a User mode handler, it must be passed somewhere other than the *Status* register.

For a floating-point operation that sets only unenabled *Cause* bits, no floating-point exception occurs and the default result defined by IEEE 754 is stored. In this case, the exceptions that were caused by the immediately previous floating-point operation can be determined by reading the *Cause* field.

Flag Bits

The *Flag* bits are cumulative and indicate the exceptions that were raised by the operations that were executed since the bits were explicitly reset. *Flag* bits are set to 1 if an IEEE 754 exception is raised, otherwise they remain unchanged. The *Flag* bits are never cleared as a side effect of floating-point operations; however, they can be set or cleared by writing a new value into the *Status* register, using a *CTC1* instruction.

When a floating-point exception is trapped, the flag bits are not set by the hardware; floating-point exception software is responsible for setting these bits before invoking a user handler.

Control/Status Register Rounding Mode Control Bits

Bits 1 and 0 in the *Control/Status* register constitute the *Rounding Mode (RM)* field.

As shown in Table 10-5, these bits specify the rounding mode that CP1 uses for all floating-point operations.

Table 10-5. Rounding Mode Bit Decoding

Rounding ModeRM (1:0)	Mnemonic	Description
0	RN	Round result to nearest representable value; round to value with least-significant bit 0 when the two nearest representable values are equally near.
1	RZ	Round toward 0: round to value closest to and not greater in magnitude than the infinitely precise result.
2	RP	Round toward +∞: round to value closest to and not less than the infinitely precise result.
3	RM	Round toward –∞: round to value closest to and not greater than the infinitely precise result.

10.2.4 Accessing the FP Control and Implementation/Revision Registers

The *Control/Status* and the *Implementation/Revision* registers are read by a Move Control From Coprocessor 1 (*CFC1*) instruction.

The bits in the *Control/Status* register can be set or cleared by writing to the register using a Move Control To Coprocessor 1 (*CTC1*) instruction. The *Implementation/Revision* register is a read-only register. There are no pipeline hazards (between any instructions) associated with floating-point control registers.

10.3 Floating-Point Formats

CP1 performs both 32-bit (single-precision) and 64-bit (double-precision) IEEE standard floating-point operations. The 32-bit single-precision format has a 24-bit signed-magnitude fraction field *(f+s)* and an 8-bit exponent *(e)*, as shown in Figure 10-5.

31	30 ... 23	22 ... 0
s Sign	e Exponent	f Fraction
1	8	23

Figure 10-5. Single-Precision Floating-Point Format

The 64-bit double-precision format has a 53-bit signed-magnitude fraction field *(f+s)* and an 11-bit exponent, as shown in Figure 10-6.

63	62 ... 52	51 ... 0
s Sign	e Exponent	f Fraction
1	11	52

Figure 10-6. Double-Precision Floating-Point Format

As shown in the above figures, numbers in floating-point format are composed of three fields:

- sign field, s
- biased exponent, $e = E + bias$
- fraction, $f = b_1 b_2 .... b_{p-1}$

where $bias$ = 127, p = 24 in single precision,

$bias$ = 1023, p = 53 in double precision

The range of the unbiased exponent *E* includes every integer between the two values E_{min} and E_{max} inclusive, together with two other reserved values:

- $E_{min} - 1$ (to encode 0 and denormalized numbers)
- $E_{max} + 1$ (to encode ∞ and NaNs [Not a Number])

For single-and double-precision formats, each representable nonzero numerical value has just one encoding uniquely.

For single-and double-precision formats, the value of a number, *v*, is determined by the equations shown in Table 10-6.

Table 10-6. Equations for Calculating Values in Single and Double-Precision Floating-Point Format

Equation	Condition
v = NaN	$E = E_{max}+1$ and $f \neq 0$, regardless of s
$v = (-1)^s\infty$	$E = E_{max}+1$ and $f = 0$
$v = (-1)^s 2^E (1.f)$	$E_{min} \leq E \leq Emax$
$v = (-1)^s 2^{Emin} (0.f)$	$E = E_{min}-1$ and $f \neq 0$
$v = (-1)^s 0$	$E = E_{min}-1$ and $f = 0$

For all floating-point formats, if *v* is NaN, the most-significant bit of *f* determines whether the value is a signaling or quiet NaN: *v* is a signaling NaN if the most-significant bit of *f* is set, otherwise, *v* is a quiet NaN.

Table 10-7 defines the values for the format parameters; minimum and maximum floating-point values are given in Table 10-8.

Table 10-7. Floating-Point Format Parameter Values

Parameter	Format	
	Single	Double
E_{max}	+127	+1023
E_{min}	–126	–1022
Exponent *bias*	+127	+1023
Exponent width in bits	8	11
Integer bit	hidden	hidden
Fraction width in bits	23†	52†
Format width in bits	32	64

† Excluding the sign bit.

Table 10-8. Minimum and Maximum Floating-Point Values

Type	Value
Float Minimum	$1.40129846e^{-45}$
Float Minimum Norm	$1.17549435e^{-38}$
Float Maximum	$3.40282347e^{+38}$
Double Minimum	$4.9406564584124654e^{-324}$
Double Minimum Norm	$2.2250738585072014e^{-308}$
Double Maximum	$1.7976931348623157e^{+308}$

10.4 Binary Fixed-Point Format

Binary fixed-point values are held in 2's complement format. Unsigned fixed-point values are not directly provided by the floating-point instruction set. Figure 10-7 illustrates binary word fixed-point format and Figure 10-8 illustrates binary long fixed-point format; Table 10-9 lists the binary fixed-point format fields.

31	30 … 0
Sign	Integer
1	31

Figure 10-7. Binary Word Fixed-Point Format

63	62 … 0
Sign	Integer
1	63

Figure 10-8. Binary Long Fixed-Point Format

Field assignments of the binary fixed-point format are:

Table 10-9. Binary Fixed-Point Format Fields

Field	Description
sign	sign bit
integer	integer value (2's complement)

10.5 Floating-Point Instruction Set Summary

Each instruction is 32 bits long, and aligned on a word boundary. This section describes the overview of instructions for floating-point unit. A detailed description of each instruction is provided in Appendix D.

10.5.1 Load, Store and Move Instructions (Table 10-10)

Load and Store instructions move data between memory and FPU general purpose registers(FGR), and Move instructions move data directly between CPU and FPU general purpose registers(FGR). These instructions are not perform format conversions and therefore never cause floating-point exceptions. The instruction immediately following a load can use the contents of the loaded register. However, in such case the hardware interlocks, requiring additional real cycles. Thus, the scheduling of load delay slots is required to avoid the interlocking.

Table 10-10. FPU Instruction Set (Optional): Load, Move and Store Instruction

Instruction	Description	Note
LWC1	Load Word to FPU (coprocessor 1)	MIPS I
SWC1	Store Word from FPU (coprocessor 1)	MIPS I
MTC1	Move Word to FPU (coprocessor 1)	MIPS I
MFC1	Move Word from FPU (coprocessor 1)	MIPS I
CTC1	Move Control Word to FPU (coprocessor 1)	MIPS I
CFC1	Move Control Word from FPU (coprocessor 1)	MIPS I
LDC1	Load Doubleword to FPU (coprocessor1)	MIPS II
SDC1	Store Doubleword from FPU (coprocessor1)	MIPS II
DMTC1	Move Doubleword to FPU (coprocessor1)	MIPS III
DMFC1	Move Doubleword from FPU (coprocessor1)	MIPS III

10.5.2 Conversion Instructions (Table 10-11)

Conversion instructions perform conversion operations between the various data formats.

Table 10-11. FPU Instruction Set(Optional): Conversion Instruction

Instruction	Description	Note
CVT.S.fmt	Floating-Point Convert to Single FP Format	MIPS I
CVT.W.fmt	Floating-Point Convert to Word Fixed-Point Format	MIPS I
CVT.D.fmt	Floating-Point Convert to Double FP Format	MIPS I
ROUND.W.fmt	Floating-point Round to Word Fixed-Point	MIPS II
TRUNC.W.fmt	Floating-point Truncate to Word Fixed-Point	MIPS II
CEIL.W.fmt	Floating-point Ceiling Convert to Word Fixed-Point	MIPS II
FLOOR.W.fmt	Floating-point Floor Convert to Word Fixed-Point	MIPS II
CVT.L.fmt	Floating-Point Convert to Long Fixed-Point Format	MIPS III
ROUND.L.fmt	Floating-point Round to Long Fixed-Point	MIPS III
TRUNC.L.fmt	Floating-point Truncate to Long Fixed-Point	MIPS III
CEIL.L.fmt	Floating-point Ceiling Convert to Long Fixed-Point	MIPS III
FLOOR.L.fmt	Floating-point Floor Convert to Long Fixed-Point	MIPS III

10.5.3 Computational Instructions (Table 10-12)

Computational instructions perform arithmetic operations on floating-point values in the FPU registers. These are two categories of computational instructions:

- 3-Operand Register-Type instructions, which perform floating-point addition, subtraction multiplication, and division operations
- 2-Operand Register-Type instructions, which perform floating-point abusolute value, move, negate, and square root operations.

Table 10-12. FPU Instruction Set(Optional): Computational Instruction

Instruction	Description	Note
ADD.fmt	Floating-point Add	MIPS I
SUB.fmt	Floating-point Subtract	MIPS I
MUL.fmt	Floating-point Multiply	MIPS I
DIV.fmt	Floating-point Divide	MIPS I
ABS.fmt	Floating-point Absolute Value	MIPS I
MOV.fmt	Floating-point Move	MIPS I
NEG.fmt	Floating-point Negate	MIPS I
SQRT.fmt	Floating-point Square root	MIPS II

10.5.4 Compare and Branch Instructions (Table 10-13)

Compare instructions perform comparisons of the contents of registers and set a conditional bit based on the results. Branch on FPU Condition instructions perform a branch to the specified target if the specified coprocessor condition is met.

Table 10-13. FPU Instruction Set(Optional): Compare and Branch Instruction

Instruction	Description	Note
C.cond.fmt	Floating-point Compare	MIPS I
BC1T	Branch on FPU True	MIPS I
BC1F	Branch on FPU False	MIPS I

11. Floating-Point Exception (Option)

This chapter describes FPU floating-point exceptions, including FPU exception types, exception trap processing, exception flags, saving and restoring state when handling an exception, and trap handlers for IEEE Standard 754 exceptions.

A floating-point exception occurs whenever the FPU cannot handle either the operands or the results of a floating-point operation in its normal way. The FPU responds by generating an exception to initiate a software trap or by setting a status flag.

11.1 Introduction

This chapter describes floating-point exceptions, including FPU exception type, exception trap processing, exception flags, saving and restoring state when handling an exception, and trap handlers for IEEE Standard 754 exceptions.

11.2 Exception Types

The FP Control/Status register described in Chapter 10 contains an Enable bit for each exception type; exception Enable bits determine whether an exception will cause the FPU to initiate a trap or set a status flag.

- If a trap is taken, the FPU remains in the state found at the beginning of the operation and a software exception handling routine executes.
- If no trap is taken, an appropriate value is written into the FPU destination register and execution continues.

The FPU supports the five IEEE Standard 754 exceptions:

- Inexact (I)
- Underflow (U)
- Overflow (O)
- Division by Zero (Z)
- Invalid Operation (V)

Cause bits, Enables, and Flag bits (status flags) are used.

The FPU adds a sixth exception type, Unimplemented Operation (E). This exception indicates the use of a software implementation. The Unimplemented Operation exception has no Enable or Flag bit; whenever this exception occurs, an unimplemented exception trap is taken.

Figure 11-1 shows the Control/Status register bits that support exceptions.

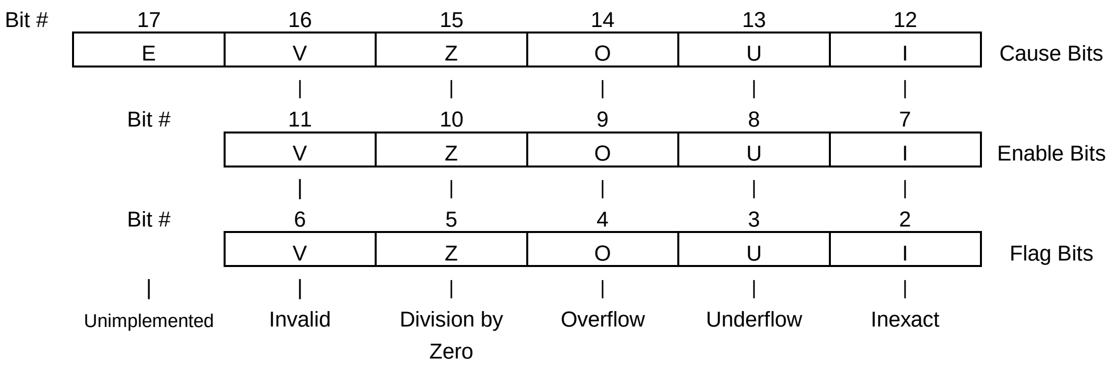


Figure 11-1. Control/Status Register Exception/Flag/Trap/Enable Bits

11.3 Exception Trap Processing

When a floating-point exception trap is taken, the Cause register indicates the floating-point coprocessor is the cause of the exception trap.

The Floating-Point Exception (FPE) code is used, and the Cause bits of the floating-point Control/Status register indicate the reason for the floating-point exception. These bits are, in effect, an extension of the system coprocessor Cause register.

11.4 Flags

A Flag bit is provided for each IEEE exception. This Flag bit is set to a 1 on the assertion of its corresponding exception, without corresponding exception trap signaled.

The Flag bit is reset by writing a new value into the Status register; flags can be saved and restored by software either individually or as a group.

When no exception trap is signaled, floating-point coprocessor takes a default action, providing a substitute value for the exception-causing result of the floating-point operation. The particular default action taken depends upon the type of exception. Table 11-1 lists the default action taken by the FPU for each of the IEEE exceptions.

Table 11-1. Default FPU Exception Actions

Field	Description	Rounding Mode	Default action
I	Inexact exception	Any	Supply a rounded result
U	Underflow exception	RN	Modify underflow values to 0 with the sign of the intermediate result
		RZ	Modify underflow values to 0 with the sign of the intermediate result
		RP	Modify positive underflows to the format's smallest positive finite number; modify negative underflows to –0.
		RM	Modify negative underflows to the format's smallest negative finite number; modify positive underflows to 0.
O	Overflow exception	RN	Modify overflow values to ∞ with the sign of the intermediate result
		RZ	Modify overflow values to the format's largest finite number with the sign of the intermediate result
		RP	Modify negative overflows to the format's most negative finite number; modify positive overflows to +∞
		RM	Modify positive overflows to the format's largest finite number; modify negative overflows to –∞
Z	Division by zero	Any	Supply a properly signed ∞
V	Invalid operation	Any	Supply $2^{31}-1$ result (Word Fixed-Point); Supply $2^{67}-1$ result (Long Fixed-Point); Otherwise supply a quiet Not a Number

The FPU detects the eight exception causes internally. When the FPU encounters one of these unusual situations, it causes either an IEEE exception or an Unimplemented Operation exception (E).

Table 11-2 lists the exception-causing situations and contrasts the behavior of the FPU with the requirements of the IEEE Standard 754.

Table 11-2. FPU Exception-Causing Conditions

FPA Internal Result	IEEE Standard 754	Trap Enable	Trap Disable	Notes
Inexact result	I	I	I	Loss of accuracy
Exponent overflow	O, I (*1)	O, I	O, I	Normalized exponent > E_{max}
Division by zero	Z	Z	Z	Zero is (exponent=E_{min} –1, mantissa=0)
Overflow on convert to Integer	V	V (*2)	V (*2)	Source out of integer range, ∞, NaN
Signaling NaN source	V	V	V	
Invalid operation	V	V	V	0/0, etc.
Exponent underflow	U	E	UI (*3)	Normalized exponent < E_{min}
Denormalized or QNaN	None	E	E	Denormalized is (exponent=E_{min} –1 and mantissa <> 0)

(*1) The IEEE Standard 754 specifies an inexact exception on overflow only if the overflow trap is disabled.

(*2) Some implementations such as TX49 trap as (E) and SW support is requred. In TX79 implementation there is NO SW support required.

(*3) Exponent underflow sets the U and I *Cause* bits if both the U and I *Enable* bits are not set and the *FS* bit is set; otherwise exponent underflow sets the E *Cause* bit.

11.5 FPU Exceptions

The following sections describe the conditions that cause the FPU to generate each of its exceptions, and details the FPU response to each exception-causing condition.

Inexact Exception (I)

The FPU generates the Inexact exception if one of the following occurs:

- the rounded result of an operation is not exact, or
- the rounded result of an operation overflows, or
- the rounded result of an operation underflows and both the Underflow and Inexact Enable bits are not set and the *FS* bit is set.

Trap Enabled Results: If Inexact exception traps are enabled, the result register is not modified and the source registers are preserved.

Trap Disabled Results: The rounded or overflowed result is delivered to the destination register if no other software trap occurs.

Invalid Operation Exception (V)

Floating-Point format operation

The Invalid Operation exception is signaled if one or both of the operands are invalid for an implemented operation. When the exception occurs without a trap, the MIPS ISA defines the result as a quiet Not a Number (QNaN) for Floating-Point format. The invalid operations are:

- Addition or subtraction: magnitude subtraction of infinities, such as: $(+\infty)+(-\infty)$ or $(-\infty)-(-\infty)$
- Multiplication: 0 times ∞, with any signs
- Division: 0/0, or ∞/∞, with any signs
- Comparison of predicates involving '<' or '>' without '?', when the operands are unordered*
- Any arithmetic operation, when one or both operands is a signaling NaN. A move (MOV) operation is not considered to be an arithmetic operation, but absolute value (ABS) and negate (NEG) are considered to be arithmetic operations.
- Comparison or Convertion From Floating-point Format on a signaling NaN.
- Square root: $\sqrt{x}$, where x is less than zero.

Software can simulate the Invalid Operation exception for other operations that are invalid for the given source operands. Examples of these operations include IEEE Standard 754-specified functions implemented in software, such as Remainder: *x* REM *y*, where *y* is 0 or *x* is infinite; conversion of a floating-point number to a decimal format whose value causes an overflow, is infinity, or is NaN; and transcendental functions, such as ln (–5) or $\cos^{-1}$ (3). Refer to Appendix D for examples or for routines to handle these cases.

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: A quiet NaN is delivered to the destination register if no other software trap occurs.

Conversion to Integer format

The Invalid Operation exception is also raised when the source operand is an Infinity (∞) or NaN, or the correctly rounded integer result is outside of the representable range.

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disable Results: The result value $2^{31}-1$ (for Word Fixed-Point) or $2^{63}-1$ (for Long Fixed-Point) is delivered to the destination register if no other software trap occurs.

---

* '<', '>' and '?' are the notation in IEEE std 754.
'?' means 'unordered.' See Compare instruction in Appendix D.

Division-by-Zero Exception (Z)

The Division-by-Zero exception is signaled on an implemented divide operation if the divisor is zero and the dividend is a finite nonzero number. Software can simulate this exception for other operations that produce a signed infinity, such as In (0), sec (π/2), csc (0), or 0^{-1}

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: The result, when no trap occurs, is a correctly signed infinity.

Overflow Exception (O)

The Overflow exception is signaled when the magnitude of the rounded floating-point result, with an unbounded exponent range, is larger than the largest finite number of the destination format. (This exception also signals an Inexact exception.)

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: The result, when no trap occurs, is determined by the rounding mode and the sign of the intermediate result (see Table 11-3).

Table 11-3. Values of Overflow Results

Denormalized Result	Flushed result Rounding Mode			
	RN	RZ	RP	RM
Positive	+∞	+Emax	+∞	+Emax
Negative	–∞	–Emax	–Emax	–∞

Underflow Exception (U)

Two related events contribute to the Underflow exception:

- creation of a tiny nonzero result between $\pm 2^{Emin}$ which can cause some later exception because it is so tiny
- extraordinary loss of accuracy during the approximation of such tiny numbers by denormalized numbers.

IEEE Standard 754 allows a variety of ways to detect these events, but requires they be detected the same way for all operations.

Tininess can be detected by one of the following methods:

- after rounding (when a nonzero result, computed as though the exponent range were unbounded, would lie strictly between $\pm 2^{Emin}$)
- before rounding (when a nonzero result, computed as though the exponent range and the precision were unbounded, would lie strictly between $\pm 2^{Emin}$).

The MIPS architecture requires that tininess be detected after rounding.

Loss of accuracy can be detected by one of the following methods:

- denormalization loss (when the delivered result differs from what would have been computed if the exponent range were unbounded)
- inexact result (when the delivered result differs from what would have been computed if the exponent range and precision were both unbounded).

The MIPS architecture requires that loss of accuracy be detected as an inexact result.

Trap Enabled Results: If Underflow or Inexact traps are enabled, or if the *FS* bit is not set, then an Unimplemented exception (E) is generated, and the result register is not modified and the source registers are preserved.

Trap Disabled Results: If Underflow and Inexact traps are not enabled and the *FS* bit is set, the result is determined by the rounding mode and the sign of the intermediate result (See Table 10-4).

Unimplemented Instruction Exception (E)

Any attempt to execute an instruction with an operation code or format code that has been reserved for future definition sets the *Unimplemented* bit in the *Cause* field in the FPU *Control/Status* register and traps. The operand and destination registers remain undisturbed and the instruction is emulated in software. Any of the IEEE Standard 754 exceptions can arise from the emulated operation, and these exceptions are simulated.

The Unimplemented Instruction exception can also be signaled when unusual operands or result conditions are detected that the implemented hardware cannot handle properly. These include:

- Denormalized operand, except for Compare instruction
- Quiet Not a Number operand, except for Compare instruction
- Denormalized result or Underflow, when either Underflow or Inexact *Enable* bit is set or the *FS* bit is not set.
- Reserved opcodes
- Unimplemented formats
- Operations which are invalid for their format (for instance, CVT.S.S)

NOTE: Denormalized and NaN operands are only trapped if the instruction is a convert or a computational operation. A move opration does not trap if their operands are either denormalized or NaNs.

The use of this exception for such conditions is optional; most of these conditions are newly developed and are not expected to be widely used in early implementations. Loopholes are provided in the architecture so that these conditions can be implemented with assistance provided by software, maintaining full compatibility with the IEEE Standard 754.

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: This trap cannot be disabled.

11.6 Saving and Restoring State

Sixteen doubleword† coprocessor load or store operations save or restore the coprocessor floating-point register state in memory. The remainder of control and status information can be saved or restored through *CFC1/CTC1* instructions, and saving and restoring the processor registers. Normally, the *Control/Status* register is saved first and restored last.

When state is restored, state information in the *Control/Status* register indicates the exceptions that are pending. Writing a zero value to the *Cause* field of *Control/Status* register clears all pending exceptions, permitting normal processing to restart after the floating-point register state is restored.

11.7 Trap Handlers for IEEE Standard 754 Exceptions

The IEEE Standard 754 strongly recommends that users be allowed to specify a trap handler for any of the five standard exceptions so that a software subroutine can return a value to be used in stead of the exceptional operation's result; the trap handler can either compute or specify a substitute result to be placed in the destination register of the operation.

By retrieving an instruction using the processor *Exception Program Counter* (*EPC*) register, the trap handler determines:

- exceptions occurred during the operation
- the operation being performed
- the destination format

On Overflow or Underflow exceptions (except for conversions), and on Inexact exceptions, the trap handler gains access to the correctly rounded result by decoding source register field of the instruction code and simulating the operation in software.

On Overflow or Underflow exceptions caused by a floating-point conversion, on Invalid Operation and on Division-by-Zero exceptions, the trap handler gains access to the operand values by decoding the source register field of the instruction code.

The IEEE Standard 754 recommends that, if enabled, the overflow and underflow traps take precedence over a separate inexact trap. This prioritization is accomplished in software; hardware sets the bits for both the Inexact exception and the Overflow or Underflow exception.

† 32 doublewords if the FR bit is set to 1.

12. PC Trace

This chapter describes the trace functions present on the C790.

The C790 supports real-time PC tracing. Pipeline status, target addresses of indirect jumps, and exception vectors are made available on special signals. The executed instruction sequence can be restored from signals and the source program.

The C790 also supports hardware breakpoints. The breakpoint facility is described in Chapter 13.

12.1 Real-Time PC Tracing

Trace information and non-sequential Program Counters are made available on special signal lines of the CPU.

The following trace information is made available:

- Instruction being executed in pipeline 0
- Instruction being executed in pipeline 1
- Current execution status (Normal (sequential), Branch Taken, Jump Target, Exception Target)

For Indirect jumps, the target address is also made available. For exception vectors, a code for the exception vector address is made available.

12.1.1 Classification of Branch and Jump Instructions

In this chapter, branches and jumps are classified into three categories which are direct jump, indirect jump and branch in order to explains the function of PC trace. The classification is show in Table 12-1.

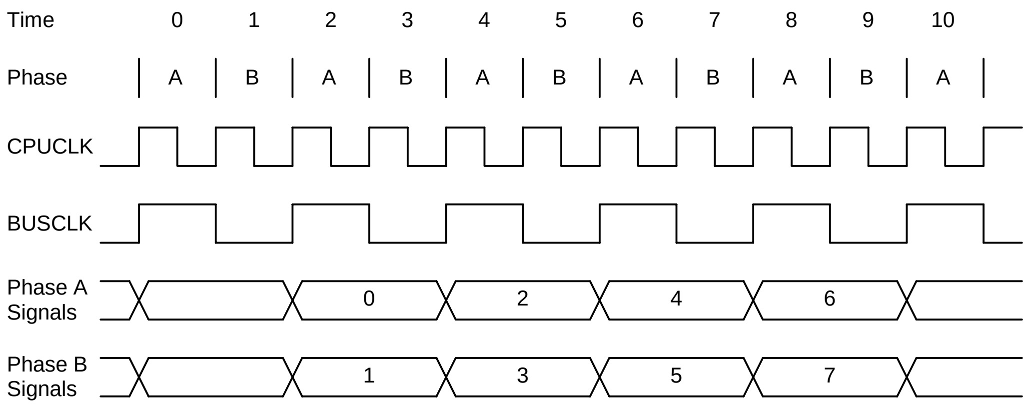
Table 12-1. Classification of Branch and Jump Instruction

Class	Instruction
Jump	Direct or Indirect Jump
Direct Jump	J or JAL Instruction
Indirect Jump	JR, JALR or ERET Instruction
Branch	Any of conditional branch Instruction

12.1.2 PC Trace Signals

All PC trace signals operate at half the C790 CPU clock frequency using the BUSCLK clock signal. Because of the half frequency operation there are pairs of signals which indicate the status of execution within the CPU pipelines. Phase A signals show the status corresponding to the *even* CPU clock cycle and Phase B signals show the status corresponding to the *odd* CPU clock cycle.

As can be seen from the following figure the execution status of the CPU pipeline during time 0 (all time references are in relation to the CPU clock) is put on the phase A signals at the next rising edge of BUSCLK during time 2. Similarly the execution status of the CPU pipeline during time 1 is put on the phase B signals.



The following signals are made available for real-time PC tracing.

- P0EXEA* (Phase A Pipeline 0 Execution Status) Output
- P1EXEA* (Phase A Pipeline 1 Execution Status) Output
- JMPA* (Phase A Jump) Output
- P0EXEB* (Phase B Pipeline 0 Execution Status) Output
- P1EXEB* (Phase B Pipeline 1 Execution Status) Output
- JMPB* (Phase B Jump) Output
- TPCE* (Target PC Enable) Output
- TPC[3:0] (Target PC Bus) Output

(1) P0EXEA* (Phase A Pipeline 0 Execution Status) Output

P0EXEA indicates whether an instruction has completed execution without generating an exception (retired) via Pipeline 0 during phase A.

0: An instruction was retired.
1: No instruction was retired.

(2) P1EXEA* **(Phase A Pipeline 1 Execution Status)** **Output**

P1EXEA indicates whether an instruction retired via Pipeline 1 during phase A. Note if this signal is asserted at the same time as P0EXEA* then two instructions were retired simultaneously during phase A via pipelines 0 and 1 but there is no indication as to which specific instruction was retired via which pipeline.

0: An instruction was retired.
1: No instruction was retired.

(3) JMPA* **(Jump Phase A)** **Output**

A jump was retired during phase A or a conditional branch instruction was retired and the branch was taken during phase A. Note that exceptions do not assert this signal.

0: Jump or conditional branch instruction was retired.
1: No Jump or conditional branch instruction was retired.

(4) P0EXEB* **(Phase B Pipeline 0 Execution Status)** **Output**

P0EXEB indicates whether an instruction retired via Pipeline 0 during phase B.

0: An instruction was retired.
1: No instruction was retired.

(5) P1EXEB* **(Phase B Pipeline 1 Execution Status)** **Output**

P1EXEB indicates whether an instruction retired via Pipeline 1 during phase B. Note if this signal is asserted at the same time as P0EXEB* then two instructions were retired simultaneously during phase B via pipelines 0 and 1 but there is no indication as to which specific instruction was retired via which pipeline.

0: An instruction was retired.
1: No instruction was retired.

(6) JMPB* **(Jump Phase B)** **Output**

A jump was retired during phase B or a conditional branch instruction was retired and the branch was taken during phase B. Note that exceptions do not assert this signal.

0: Jump or conditional branch instruction was retired.
1: No Jump or conditional branch instruction was retired.

(7) TPCE* **(Target PC Enable)** **Output**

When this signal is asserted the TPC bus indicates the type of target PC that will be made available.

0: TPC bus indicates type of target PC.
1: TPC bus has either the target PC or the exception vector address code or has no information.

The normal sequence of operation for the TPCE* and the TPC[3:0] signals is as follows: First TPCE* is asserted and simultaneously TPC[3:0] contains information about the type of the target PC (non-sequential PC). Next TPCE* is deasserted and either the target PC for indirect jumps is made available on the TPC[3:0] bus or for exceptions an exception vector address code is made available on the TPC[3:0] bus.

(8) TPC[3:0] **(Target PC)** **Output**

TPC[3:0] either indicates the type of the target PC address or the target address of indirect jump instructions or exception vector address codes.

TPC[3:0] when TPCE* is asserted

When TPCE* is asserted the type of the target PC address is made available on TPC[3:0]. Each bit of TPC[3:0] indicates a different type and multiple bits can be active at the same time.

- TPC[0]: Jump Target during Phase A

  When this signal is asserted it indicates that the target instruction of an Indirect Jump instruction (includes JR, JALR and ERET) is retired during Phase A. The target address is made available on TPC[3:0] in the next cycle if neither TPC[2] or TPC[3] are asserted simultaneously with this signal.

- TPC[1]: Exception Target during Phase A

  When this signal is asserted it indicates that the first instruction of an exception handler is retired during Phase A. The exception vector address is made available on TPC[3:0] in the next cycle if neither TPC[2] nor TPC[3] are asserted simultaneously with this signal.

- TPC[2]: Jump Target during Phase B

  When this signal is asserted it indicates that the target instruction of an Indirect Jump instruction is retired during Phase B. The target address is made available on TPC[3:0] in the next cycle.

- TPC[3]: Exception Target during Phase B

  When this signal is asserted it indicates that the first instruction of an exception handler is retired during Phase B. The exception vector address is made available on TPC[3:0] in the next cycle.

TPC[3:0] when TPCE* is deasserted

When TPCE* is not asserted TPC[3:0] can be carrying the following three type of information:

1. There is no meaningful information on TPC. This happens most of the time when the program is executing sequentially.
2. The target address is made available because in the previous cycle TPCE* was asserted and TPC[0] or TPC[2] were equal to 0. The target address starts with the least significant four bits of the target instruction address (bits[5:2]).
3. An exception vector address code is made available because in the previous cycle TPCE* was asserted and TPC[1] or TPC[3] were equal to 0. The exception vector address code are shown in Table 12-2.

Table 12-2. Exception Vector Address Codes

Exception	STATUS.BEV	STATUS.DEV	STATUS.EXL	Vector Address	Code (TPC[3:0])
Reset, NMI	x	x	x	0xBFC0 0000	8 (1000)
TLB Miss	1	x	0	0xBFC0 0200	12 (1100)
TLB Miss	0	x	0	0x8000 0000	0 (0000)
TLB Miss	1	x	1	0xBFC0 0380	15 (1111)
TLB Miss	0	x	1	0x8000 0180	3 (0011)
Debug & SIO	x	1	x	0xBFC0 0300	14 (1110)
Debug & SIO	x	0	x	0x8000 0100	2 (0010)
Performance Counter	x	1	x	0xBFC0 0280	13 (1101)
Performance Counter	x	0	x	0x8000 0080	1 (0001)
Interrupt	1	x	x	0xBFC0 0400	9 (1001)
Interrupt	0	x	x	0x8000 0200	4 (0100)
Common	1	x	x	0xBFC0 0380	15 (1111)
Common	0	x	x	0x8000 0180	3 (0011)

12.1.3 Priority of Target Addresses

The target address for an indirect jump instruction or an exception vector address code is made available on TPC[3:0]. For an indirect jump instruction it takes multiple cycles (8 BUSCLK cycles or 16 CPU clock cycles) for the complete target address to be made available on the TPC[3:0] bus. As such multiple conditions can occur simultaneously and there are certain priorities associated with putting out the target address. The rules governing what is made available on the TPC[3:0] bus are listed below:

1. If a new indirect jump instruction is retired while the target address PC for a previous indirect instruction is still being put out on TPC[3:0], the new indirect jump instruction's target PC will be signaled and start coming out on the TPC[3:0] bus and the previous target PC output will be terminated.
2. If an exception is taken while the target address PC for a previous indirect instruction is still being put out on TPC[3:0], the exception vector address code will be signaled and start coming out on the TPC[3:0] bus and the previous target PC output will be terminated

The rules are also described in the following flowchart.

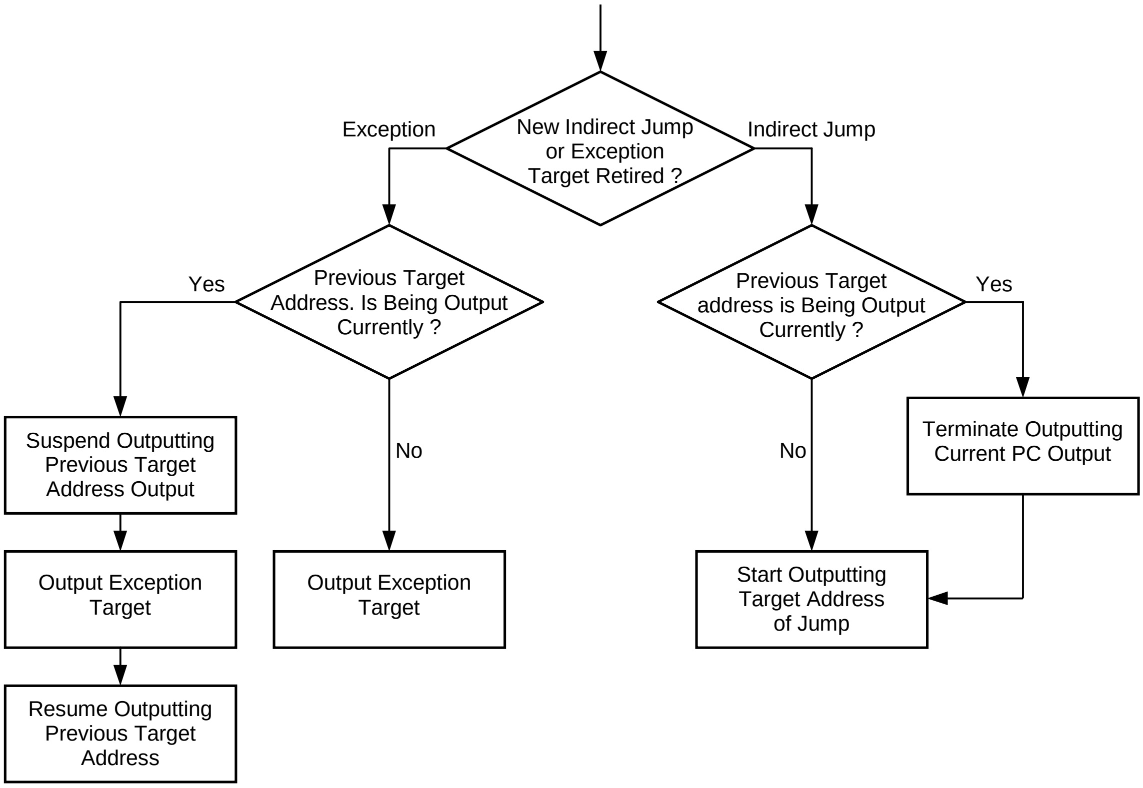


Figure 12-1. Priority of Outputting Jump or Exception Target

12.1.4 Examples of PC Tracing

The following sections contains examples of program execution and the corresponding waveforms of the PC trace signals. Note that when two instructions are retired simultaneously, just for the sake of illustration, it is indicated which instruction is executed in which pipeline. In reality, in this case, it is not known which instruction is retired from which pipeline.

12.1.4.1 Sequential Execution

This is an example of sequential program execution. The program fragment is as follows:

```
mul
add
sub
lw r1
add
sub ,,r1
add
add
```

The PC trace signals for the program fragment are shown below:

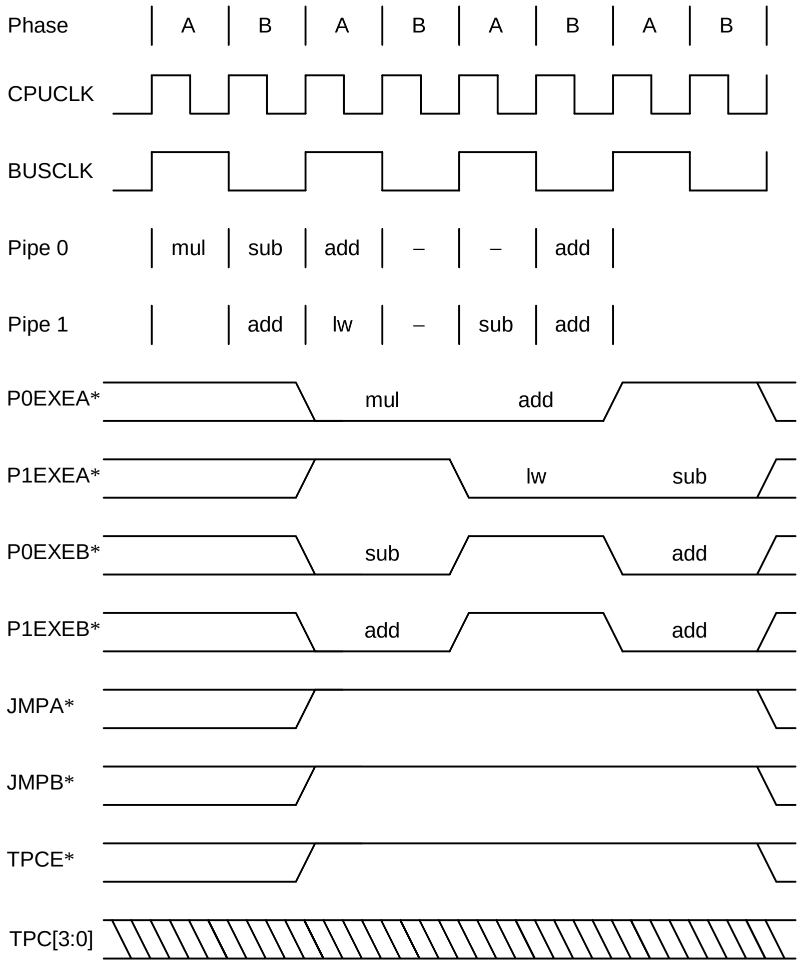


Figure 12-2. Waveform for Sequential Excecution

12.1.4.2 Conditional Branch

This is an example of program with conditional branch instructions. Both the branch taken and not taken case is illustrated. The program fragment is as follows:

```
        add
        add
        beq     L0      # Not Taken
        lw
        add
        beq     L1      # Taken
        add
        ....
L1:     add
        bne     L2      # Taken
        sll
        ....
L2:     sub
        sub
```

The PC trace signals for the program fragment are shown below:

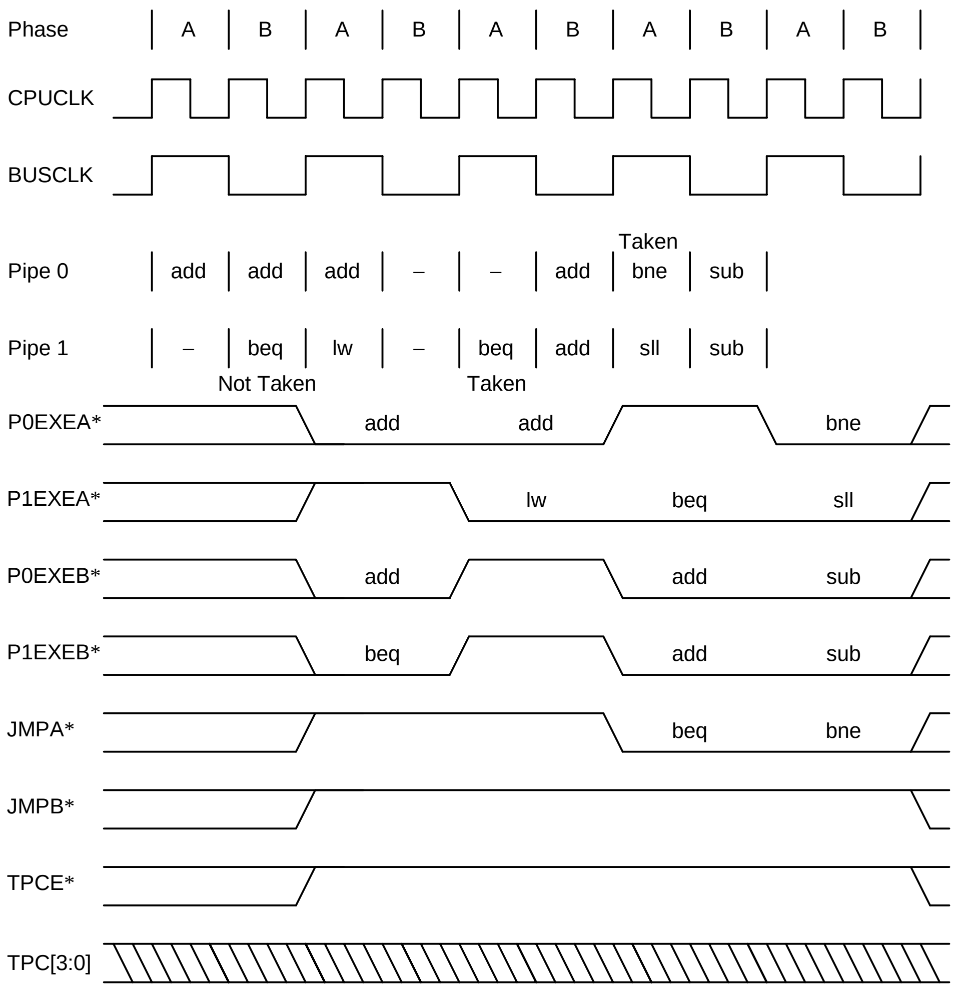


Figure 12-3. Waveform for Conditional Branch

12.1.4.3 Indirect Jump (Target in Phase A)

This is an example of program with an indirect jump instruction which is retired during phase B. The program fragment is as follows:

```
        add
        add
        jr      L1
        lw
        ....
L1:     xor
        add
        ori
        ori
        sw
        sll
        sub
        sub
```

The PC trace signals for the program fragment are shown below:

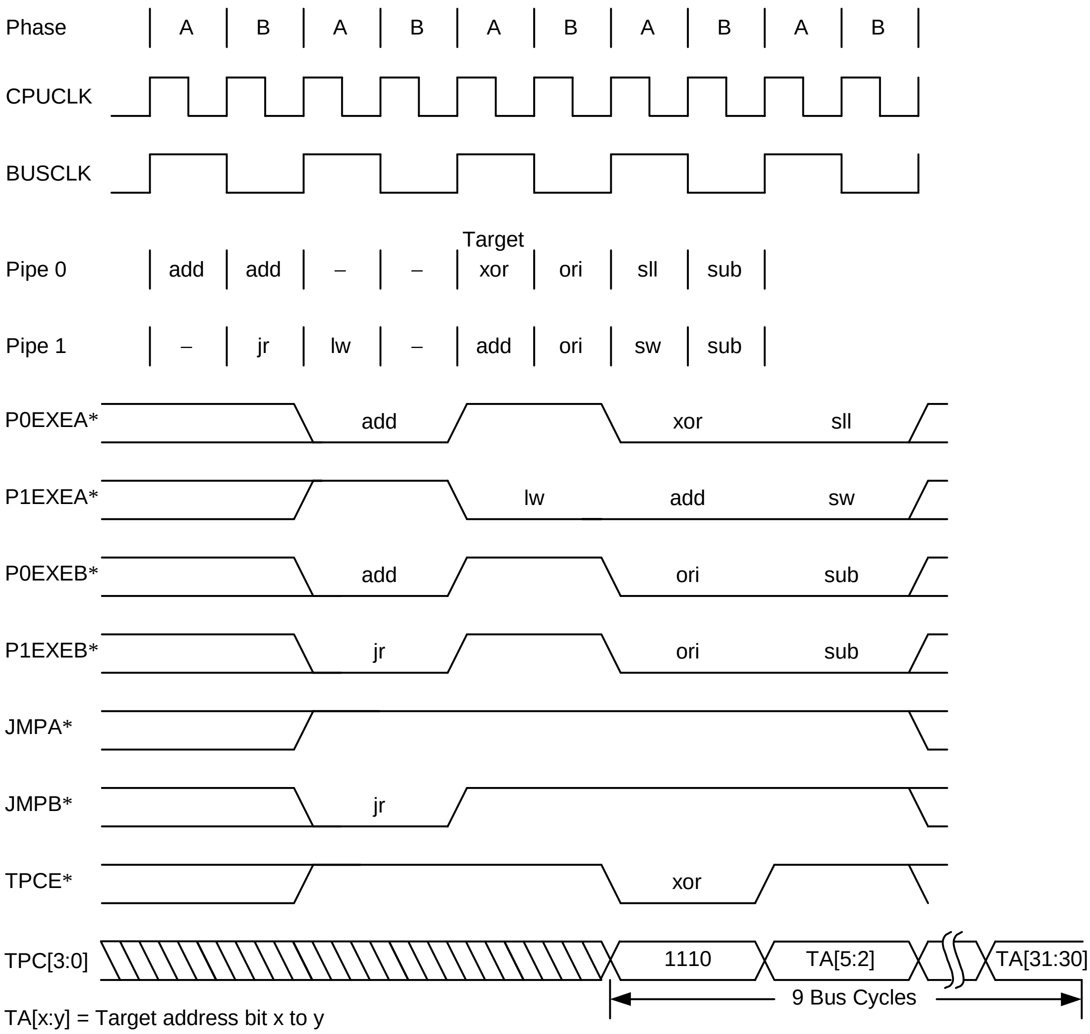


Figure 12-4. Waveform for Indirect Jump (Target in Phase A)

12.1.4.4 Indirect Jump (Target in Phase B)

This is an example of program with an indirect jump instruction which is retired during phase A. The program fragment is as follows:

```
        add
        add
        jr      L1
        lw
        ....
L1:     xor
        add
        ori
        ori
        sw
        sll
        sub
        sub
```

The PC trace signals for the program fragment are shown below:

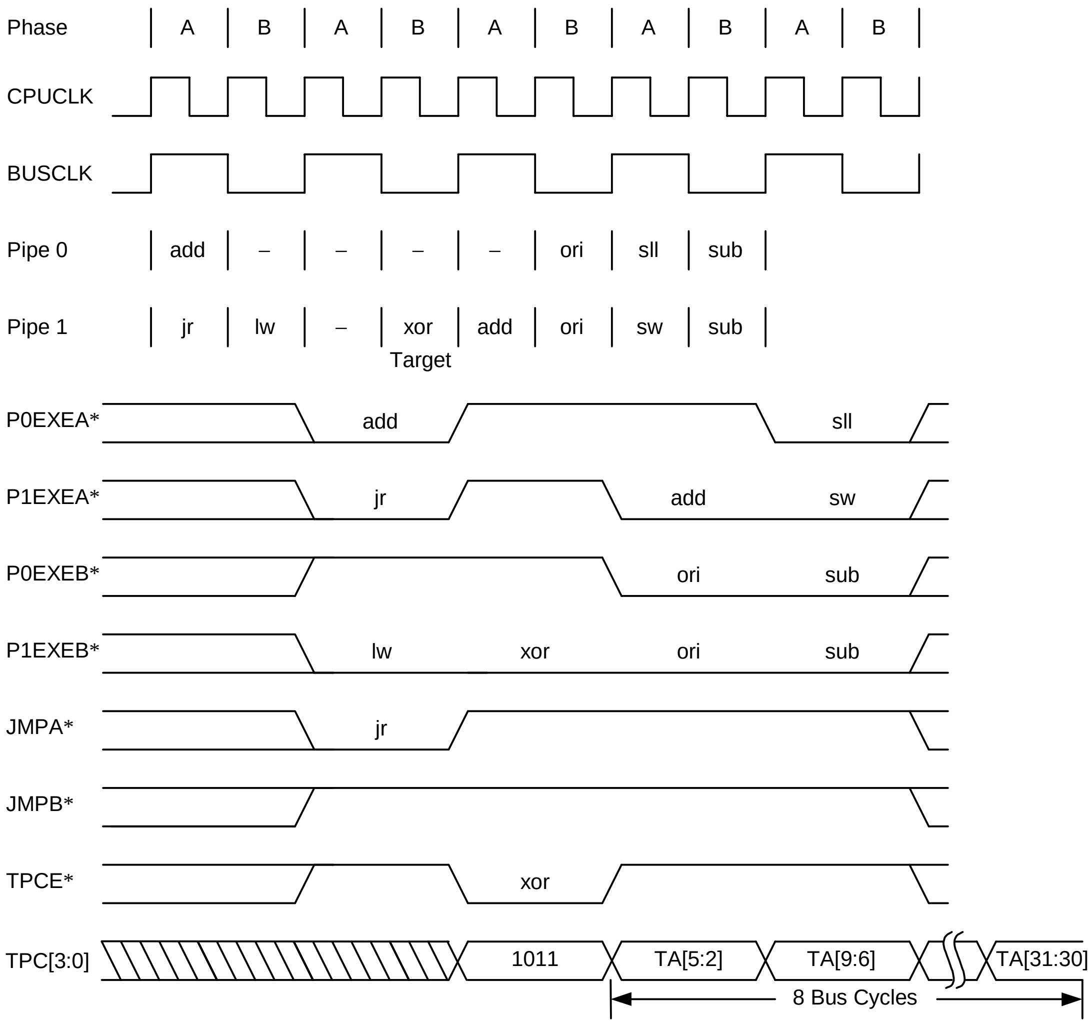


Figure 12-5. Waveform for Indirect Jump (Target in Phase B)

12.1.4.5 Indirect Jump (During Target PC Output)

This is an example of a program with two indirect jump instructions. While the target address PC associated with the first indirect jump instruction is being put out the second indirect jump instruction is retired. Thus the first target PC output is terminated and the second target PC output is signaled and then made available. The program fragment is as follows:

```
        add
        add
        jr      L1
        lw
        ....
L1:     xor
        add
        jr      L2
        add
        ....
L2      sw
        sll
        sub
        sub
```

The PC trace signals for the program fragment are shown below:

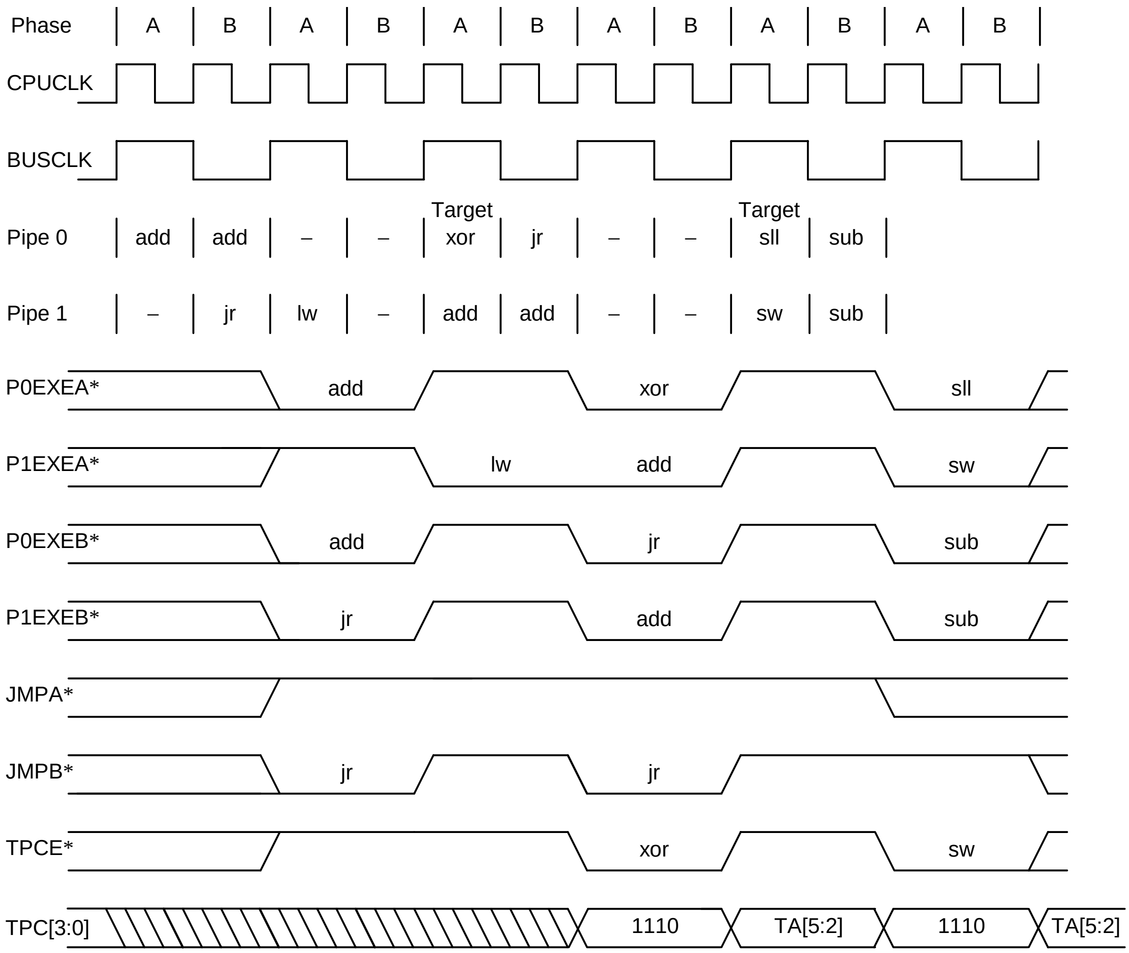


Figure 12-6. Waveform for Indirect Jump (During Target PC Output)

12.1.4.6 Exception (Target in Phase B)

This is an example of a program which generates an exception. The target instruction (first instruction of the exception handler) retires in phase B. The program fragment is shown below. The label *ExHnd* identifies the first instruction of the exception handler.

```
        add
        add
        add
        lw
        teq              # Generates exception
        ....
ExHnd:  xor
        add
        sw
        sll
        sub
        sub
```

The PC trace signals for the program fragment are shown below:

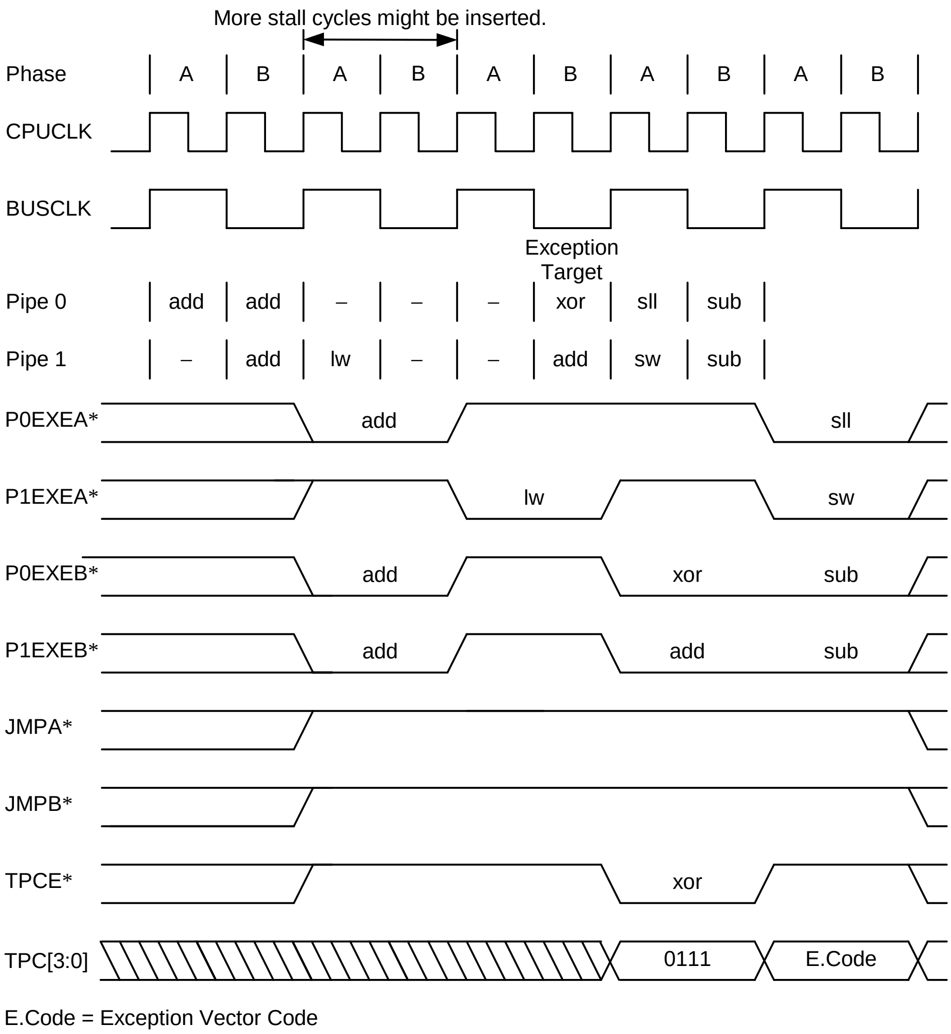


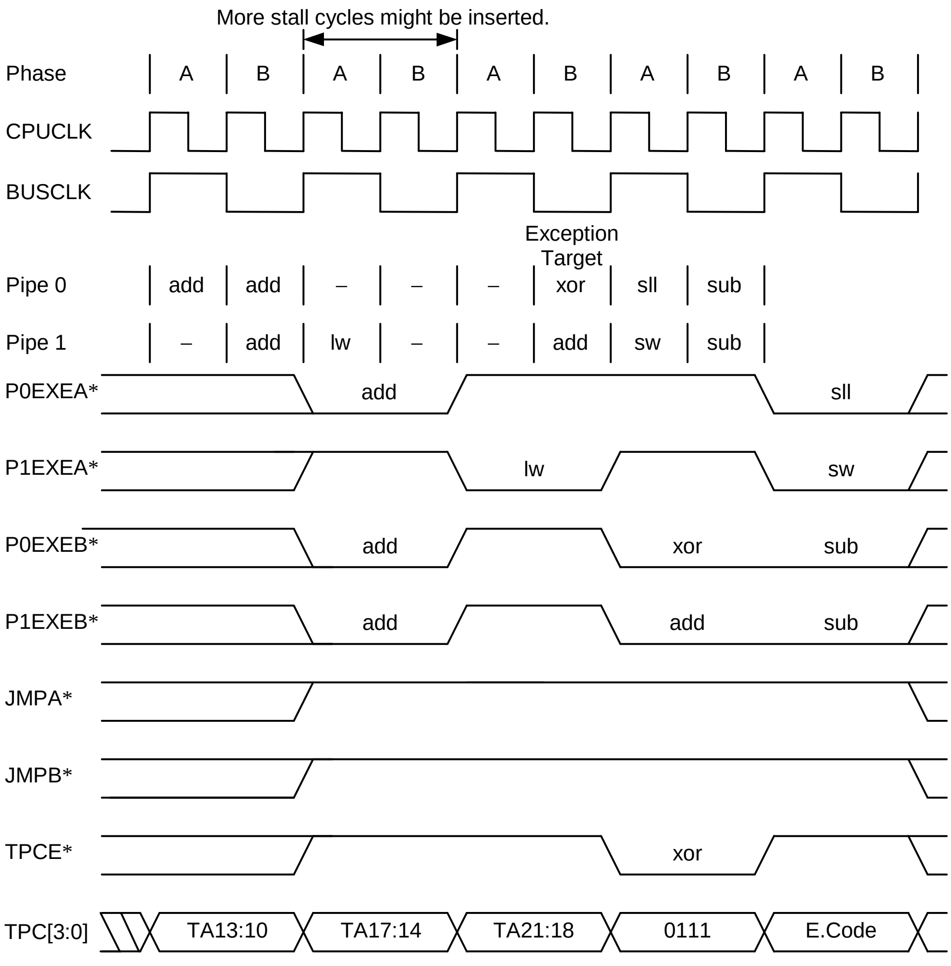
Figure 12-7. Waveform for Exception (Target in Phase B)

12.1.4.7 Exception (During Target PC Output)

This is an example of a program which generates an exception while a target PC from an earlier indirect jump instruction is being made available. The target PC output is terminated and the exception vector address code is signaled and then made available. The target instruction (first instruction of the exception handler) retires in phase B. The program fragment is shown below. The label *ExHnd* identifies the first instruction of the exception handler.

```
        add
        add
        add
        lw
        teq          # Generates exception
        ....
ExHnd:  xor
        add
        sw
        sll
        sub
        sub
```

The PC trace signals for the program fragment are shown below:



TAxx:yy = Target Address bit xx to yy
E.Code = Exception Vector Code

Figure 12-8. Waveform for Exception (During Target PC Output)

12.1.4.8 Exception Generated by Branch or Jump Instruction

This is an example of a program in which an indirect jump instruction generates an exception. As such the program jumps to the exception handler and the only thing indicated is the exception vector address code and not the jump. The target instruction (first instruction of the exception handler) retires in phase B. The program fragment is shown below. The label ExHnd identifies the first instruction of the exception handler.

```
        add
        add
        add
        lw
        jr          # Generates an exception
        nop         # Branch delay slot
        ....
ExHnd:  xor
        add
        sw
        sll
        sub
        sub
```

The PC trace signals for the program fragment are shown below:

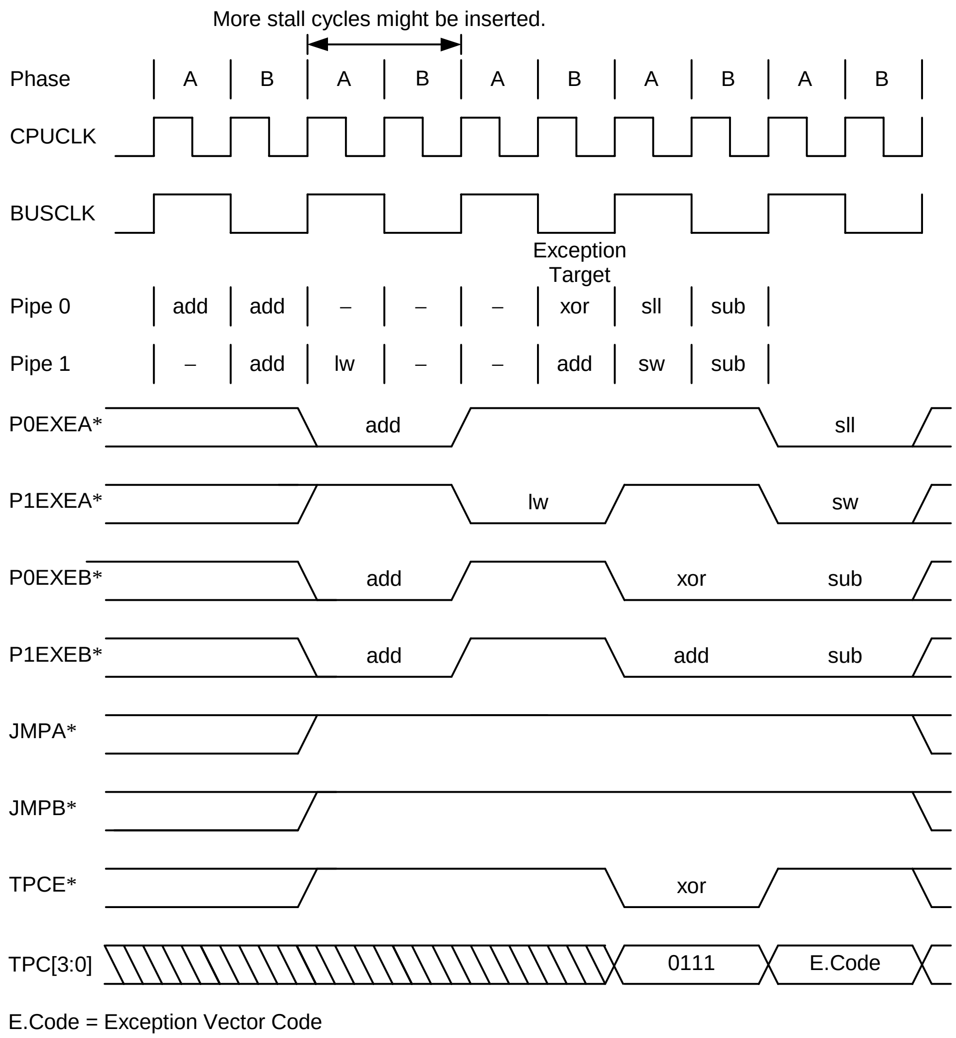


Figure 12-9. Waveform for Exception Generated by Branch or Jump Instruction

12.1.4.9 Exception Generated by Branch Delay Slot Instruction

This is an example of a program in which the branch delay slot instruction generates an exception. As such the program jumps to the exception handler and the only thing indicated is the exception vector address code and not the jump. The target instruction (first instruction of the exception handler) retires in phase B. The program fragment is shown below. The label ExHnd identifies the first instruction of the exception handler.

```
        add
        add
        add
        lw
        jr
        lw            # Generates an exception
        ....
ExHnd:  xor
        add
        sw
        sll
        sub
        sub
```

The PC trace signals for the program fragment are shown below:

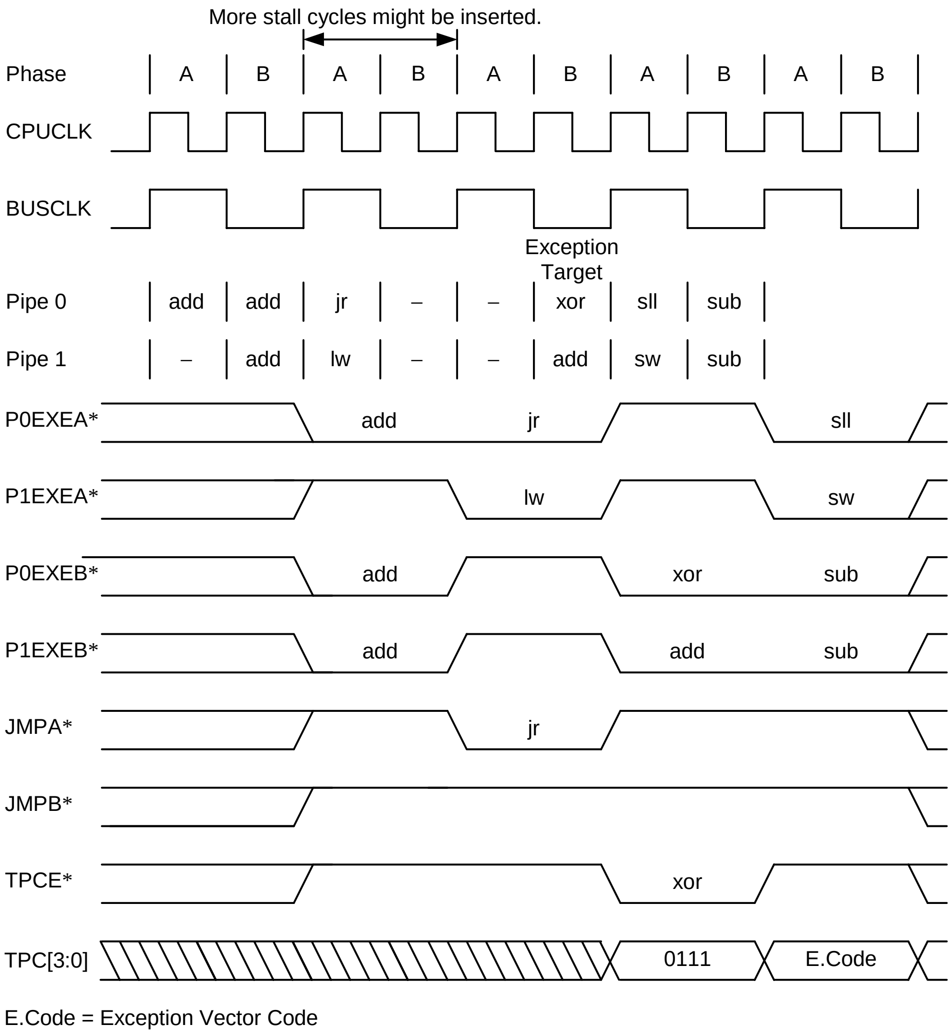


Figure 12-10. Waveform for Exception Generated by Branch Delay Slot Instruction

12.1.4.10 Exception Generated by Target Instruction

This is an example of a program in which the target instruction of an indirect jump generates an exception. As such the program jumps to the exception handler and the only thing indicated is the exception vector address code and not the jump. The target instruction (first instruction of the exception handler) retires in phase B. The program fragment is shown below. The label ExHnd identifies the first instruction of the exception handler.

```
        add
        add
        add
        lw
        jr      L1
        nop
        ....
L1:     lw            # Generates an exception
        and
        ....
ExHnd:  xor
        add
        sw
        sll
        sub
        sub
```

The PC trace signals for the program fragment are shown below:

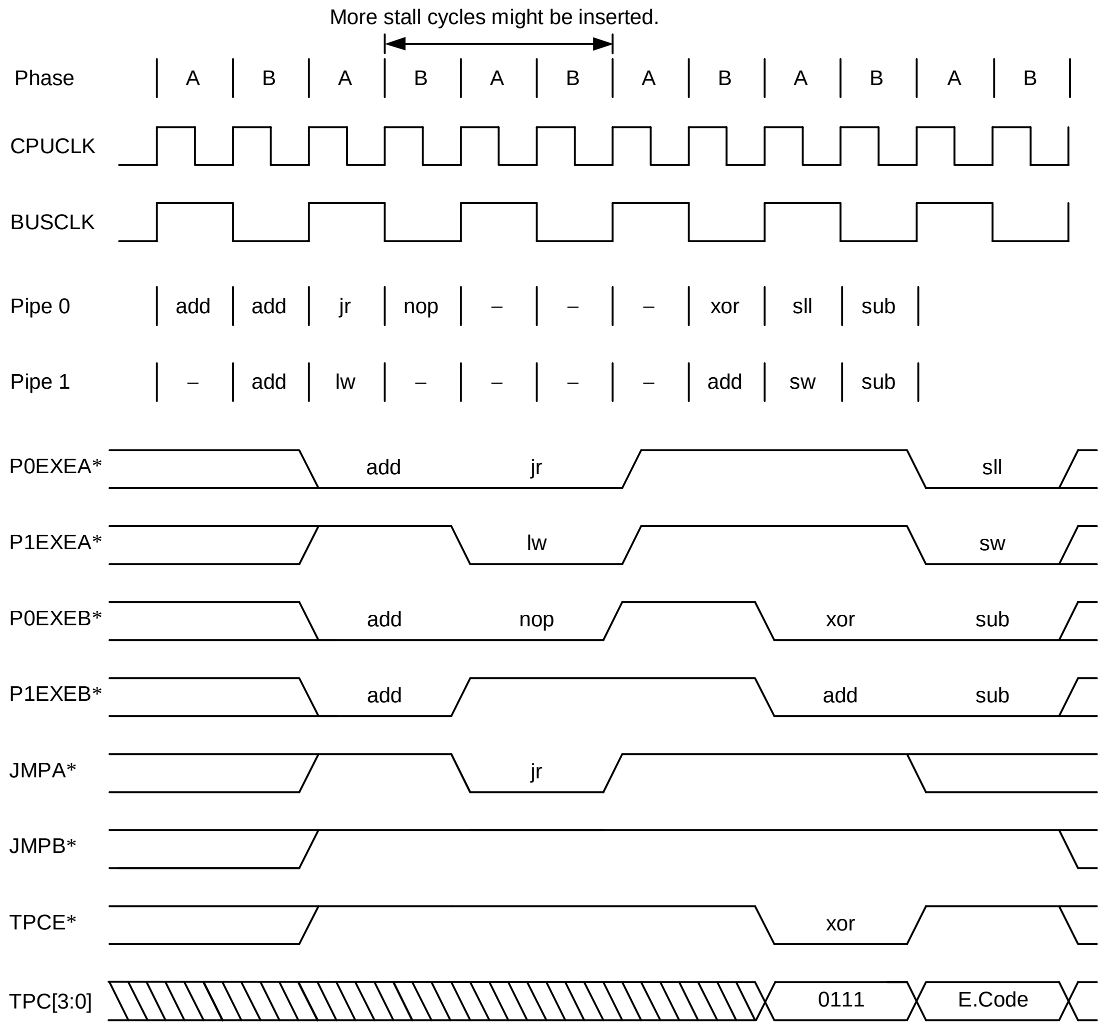


Figure 12-11. Waveform for Exception Generated by Target Instruction

12.1.4.11 Back to Back Exceptions (Case I)

This is an example of a program in which two back to back exceptions are generated. The program jumps to the first exception handler but then immediately jumps to the second exception handler. The target instruction (first instruction of the second exception handler) retires in phase A. The exception vector address code for the first handler is never made available. The program fragment is shown below. The label ExHnd1 identifies the first instruction of the first exception handler and the label ExHnd2 identifies the first instruction of the second exception handler.

```
                add
                add              # Generates the first exception
                ....
ExHnd1: xor                      # Generates the second exception
                xor
                ....
ExHnd2: sw
                sll
                sub
                sub
```

The PC trace signals for the program fragment are shown below:

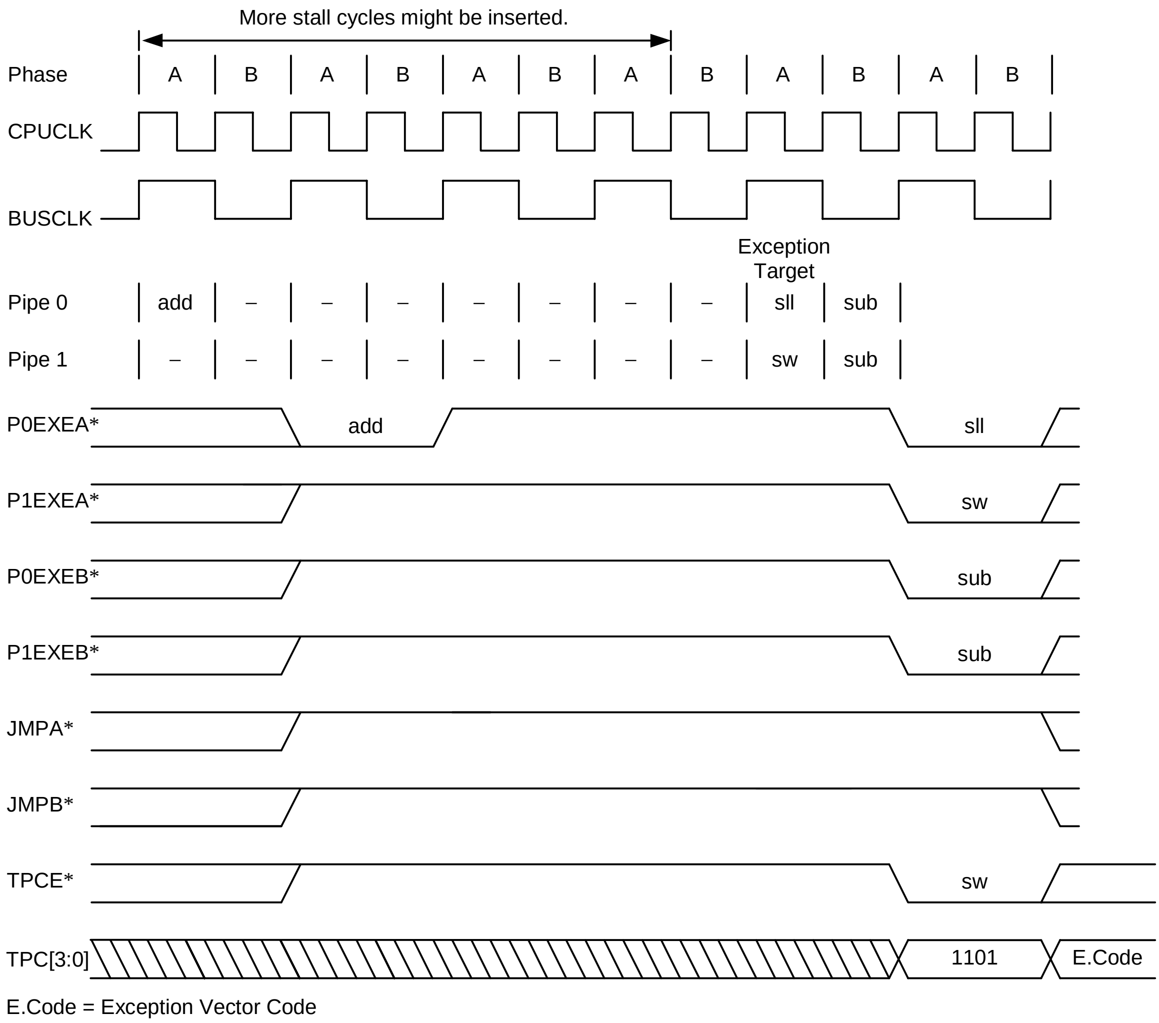


Figure 12-12. Waveform for Back to Back Exceptions (Case I)

12.1.4.12 Back to Back Exceptions (Case II)

This is an example of a program in which two (all most) back to back exceptions are generated. The program jumps to the first exception handler and then generates an exception when executing the second instruction of the exception handler. It then jumps to the second exception handler. The target instruction (first instruction of the first exception handler) retires in phase A. As compared to the case discussed above the exception vector address code for the both the handlers are made available. The program fragment is shown below. The label ExHnd1 identifies the first instruction of the first exception handler and the label ExHnd2 identifies the first instruction of the second exception handler.

```
                add
                add         # Generates the first exception
                ....
ExHnd1: xor
                xor         # Generates the second exception
                ....
ExHnd2: sw
                sll
                sub
                sub
```

The PC trace signals for the program fragment are shown below:

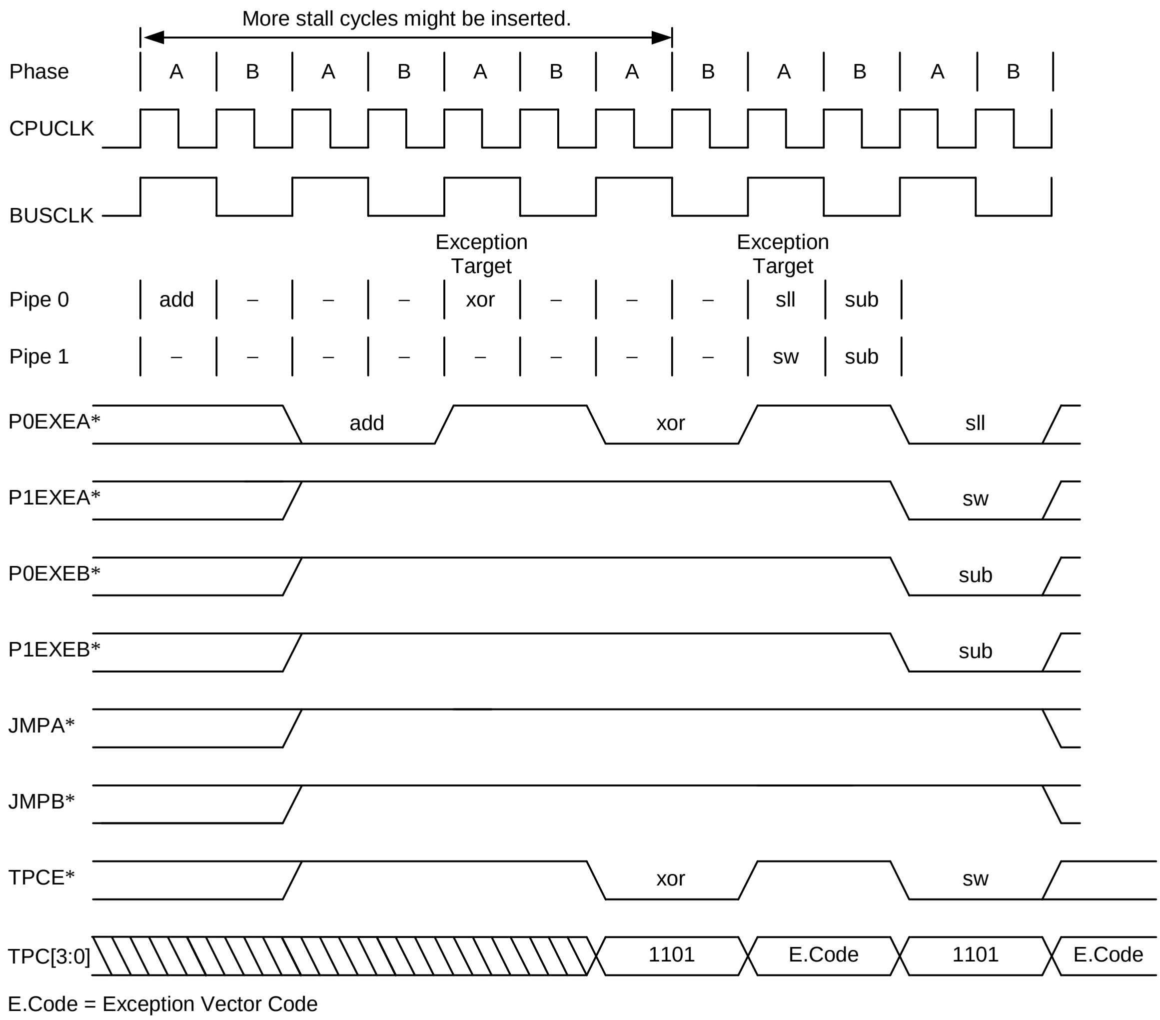


Figure 12-13. Waveform for Back to Back Exceptions (Case II)

13. Hardware Breakpoint

This chapter describes hardware breakpoint functions for debugging present on the C790.

13.1 Hardware Breakpoint

C790 provides hardware breakpoint mechanism for debugging purpose. (In this section, hardware breakpoint is sometimes referred to as “breakpoint”.) This function allows users to set a instruction breakpoint and a data address/value breakpoint with signaling the breakpoint event occurrence to external probe. The following summarizes the features of the breakpoint function.

- Provides both instruction and data breakpointing in virtual address.
- Instruction address breakpoint with address masking.
- Data breakpoint with masking. Data breakpoint can be set by the following events:

  Address with masking

  Value with masking

  Read/write
- Independent exception event control for instruction and data.
- Individual event control by processor operating mode/exception level.
- Provides a trigger signal to external probes synchronized with the breakpointing event.

Hardware breakpointing is implemented as a part of Coprocessor 0. Configuring the breakpoint is done by setting 7 Breakpoint registers by special *MTC0/MFC0* instructions. Figure 13-1 shows the basic structure of the breakpoint hardware.
Breakpoint can generate breakpoint exception which is categorized in Level2 exception, and has a dedicated exception vector. (See 5. Exception) This exception is only masked in Level2 mode, and exception generation itself can be controlled by the Breakpoint Control Register mentioned in the following section. Note that some of breakpoint exceptions are imprecise, for instance, setting value breakpoint for load instruction is basically imprecise because the load instruction may retire from the pipeline before actual acquisition of memory contents. The following summarizes imprecise cases:

- All data value breakpoint on load instruction
- Data value breakpoint on *SWC1* instruction

13.1.1 Hardware Breakpoint signal

To signal a breakpoint occurrence, the C790 activates a signal called TRIG, whenever a trigger condition is met.

- TRIG (Trigger Output) Output

This signal is asserted for two BUSCLK cycles when a trigger condition is met.

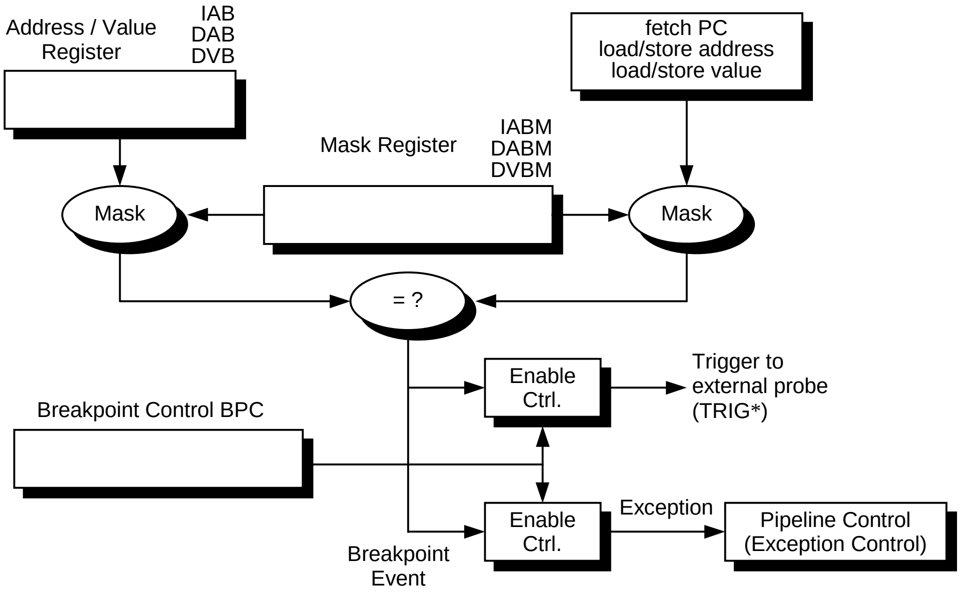


Figure 13-1. Overall Structure of Hardware Breakpoint

13.2 Breakpoint Registers

Hardware breakpoint is comprised of 3 pairs of breakpoint registers and one control register listed below. Each of breakpoint register pair includes one breakpoint value register and one breakpoint mask register.

- *Breakpoint Control* Register (BPC)
- Instruction Address Breakpoint Registers
  - *Instruction Address Breakpoint* Register (IAB)
  - *Instruction Address Breakpoint Mask* Register (IABM)
- Data Address Breakpoint Registers
  - *Data Address Breakpoint Register* (DAB)
  - *Data Address Breakpoint Mask Register* (DABM)
- Data Value Breakpoint Registers
  - *Data Value Breakpoint Register* (DVB)
  - *Data Value Breakpoint Mask Register* (DVBM)

All 7 registers are 32-bit read/write and assigned to Coprocessor0 register 24. Therefore, C790 provides extended *MTC0* instructions for accessing these registers and it is necessary to use these instructions to access these registers instead of the conventional *MTC0/MFC0* instructions. Table 13-1 and Table 13-2 summarizes the instructions for accessing the registers.

Table 13-1. Set a new value into breakpoint registers

Mnemonic	Operation
MTBPC	Move to Breakpoint Control Register
MTIAB	Move to Instruction Address Breakpoint Register
MTIABM	Move to Instruction Address Breakpoint Mask Register
MTDAB	Move to Data Address Breakpoint Register
MTDABM	Move to Data Address Breakpoint Mask Register
MTDVB	Move to Data Value Breakpoint Register
MTDVBM	Move to Data Value Breakpoint Mask Register

Table 13-2. Get the value from breakpoint registers

Mnemonic	Operation
MFBPC	Move from Breakpoint Control Register
MFIAB	Move from Instruction Address Breakpoint Register
MFIABM	Move from Instruction Address Breakpoint Mask Register
MFDAB	Move from Data Address Breakpoint Register
MFDABM	Move from Data Address Breakpoint Mask Register
MFDVB	Move from Data Value Breakpoint Register
MFDVBM	Move from Data Value Breakpoint Mask Register

13.2.1 Breakpoint Control Register (BPC)

The *BPC* register contains enable bits and status bits for controling the breakpointing of both instruction and data. This register consists of 5 parts of bit fields:

- *Breakpoint overall control* (bit [31:28])
  These bits controls the operation mode of the breakpointing.
- *Instruction breakpoint control* (bit [26:23])
  These bits specifies the processor mode that the instruction breakpoint is enabled.
- *Data breakpoint control* (bit[21:18])
  These bits specifies the processor mode that the data breakpoint is enabled.
- *Signaling Control* (bit[17:15])
  These bits controls the occurrence of breakpoint exception / trigger generation upon the breakpoint event.
- *Breakpoint Status* (bit[2:0])
  These bits indicates the type of breakpoint event. This part is used to identify which breakpoint event occurred in the breakpoint exception handler.

The following shows the detailed bitmap of BPC register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IAE	DRE	DWE	DVE	0	IUE	ISE	IKE	IXE	0	DUE	DSE	DKE	DXE	ITE	DTE	BED	0	0	0	0	0	0	0	0	0	0	0	0	DWB	DRB	IAB

Table 13-3 describes the *BPC* register fields.

Table 13-3. BPC Register Fields

Field	Bits	Description	Type	Initial Value
IAE	31	**I**nstruction **A**ddress **E**nable. This bit enables/disables instruction address breakpointing. 0: disable instruction address breakpointing 1: enable instruction address breakpointing	Read / Write	0
DRE	30	**D**ata **R**ead **E**nable. This bit enables data load address breakpointing. 0: disable breakpointing on reads 1: enable breakpointing on reads	Read / Write	0
DWE	29	**D**ata **W**rite **E**nable. This bit enables data store address breakpointing. 0: disable breakpointing on writes 1: enable breakpointing on writes	Read / Write	0
DVE	28	**D**ata **V**alue **E**nable. This bit is valid only when DRE and/or DWE are set to 1. When DVE is set to 1 data read breakpoints (DRE == 1) are further qualified by the value of the data read, and data write breakpoints (DWE == 1) are further qualified by the value of the data written. Note that data value breakpoints for data reads are imprecise. See section 13.1 (“Hardware Breakpoint”) for more details.	Read / Write	Undefined
rsvd	27	***Reserved -*** must be written as zeros by software. The processor returns zeros in these bit positions when read.	Read	0
IUE	26	**I**nstruction break - **U**ser **E**nable. This bit enables instruction address breakpointing in (standard) user mode. This bit is only valid if IAE is set to 1. 0: disable instruction address breakpointing in *User* mode 1: enable instruction address breakpointing in *User* mode	Read / Write	Undefined
ISE	25	**I**nstruction break - **S**upervisor **E**nable. This bit enables instruction address breakpointing in supervisor mode. This bit is only valid if IAE is set to 1. 0: disable instruction address breakpointing in Supervisor mode 1: enable instruction address breakpointing in Supervisor mode	Read / Write	Undefined
IKE	24	**I**nstruction break - **K**ernel **E**nable. This bit enables instruction address breakpointing in non-exception kernel mode - i.e. when both STATUS.EXL and STATUS.ERL are 0. This bit is only valid if IAE is set to 1. 0: disable instruction address breakpointing in Kernel mode 1: enable instruction address breakpointing in Kernel mode	Read / Write	Undefined
IXE	23	**I**nstruction break - E**X**L mode **E**nable. This bit enables instruction address breakpointing in exception kernel mode - i.e. when STATUS.EXL is 1 and STATUS.ERL is 0. This bit is only valid if IAE is set to 1. 0: disable instruction address breakpointing in EXL mode 1: enable instruction address breakpointing in EXL mode	Read / Write	Undefined
rsvd	22	**Reserved -** must be written as zeros by software. The processor returns zeros in these bit positions when read.	Read	0

Field	Bits	Description	Type	Initial Value
DUE	21	**D**ata break - **U**ser **E**nable. This bit enables data breakpointing in *User* mode. This bit is only valid if DWE or DRE is set to 1. 0: disable data breakpointing in *User* mode 1: enable data breakpointing in *User* mode	Read / Write	Undefined
DSE	20	**D**ata break - **S**upervisor **E**nable. This bit enables data breakpointing in *Supervisor* mode. This bit is only valid if DWE or DRE is set to 1. 0: disable data breakpointing in *Supervisor* mode 1: enable data breakpointing in *Supervisor* mode	Read / Write	Undefined
DKE	19	**D**ata break - **K**ernel **E**nable. This bit enables data breakpointing in *Kernel* mode - i.e. when both STATUS.EXL and STATUS.ERL are 0. This bit is only valid if DWE or DRE is set to 1. 0: disable data breakpointing in *Kernel* mode 1: enable data breakpointing in *Kernel* mode	Read / Write	Undefined
DXE	18	**D**ata break - E**X**L mode **E**nable. This bit enables data breakpointing in *Exception Kernel* mode - i.e. when STATUS.EXL is 1 and STATUS.ERL is 0. This bit is only valid if at least one of DRE or DWE are set to 1. 0: disable data breakpointing in *EXL* mode 1: enable data breakpointing in *EXL* mode	Read / Write	Undefined
ITE	17	**I**nstruction **T**rigger **E**nable. This bit enables the generation of the trigger signal when an instruction breakpoint occurs. 0: disable instruction breakpoint trigger 1: enable instruction breakpoint trigger	Read / Write	Undefined
DTE	16	**D**ata **T**rigger **E**nable. This bit enables the generation of the trigger signal when an data breakpoint occurs. 0: disable data breakpoint trigger 1: enable data breakpoint trigger	Read / Write	Undefined
BED	15	**B**reakpoint **E**xception **D**isable. This bit disables the entry into the debug exception handler. Note that the setting of this bit does not affect trigger signal generation. 0: enable entry into debug exception handler 1: disable entry into debug exception handler	Read / Write	Undefined
rsvd	14 - 3	**Reserved -** must be written as zeros by software. The processor returns zeros in these bit positions when read.	Read	0
DWB	2	**D**ata **W**rite **B**reakpoint. This status bit indicates whether a data breakpoint has occurred on a write or not. 0: no data breakpoint has occurred on a write 1: data breakpoint has occurred on a write	Read / Write	Undefined
DRB	1	**D**ata **R**ead **B**reakpoint. This status bit indicates whether a data breakpoint has occurred on a read or not. 0: no data breakpoint has occurred on a read 1: data breakpoint has occurred on a read	Read / Write	Undefined
IAB	0	**I**nstruction **A**ddress **B**reakpoint. This status bit indicates whether an instruction address breakpoint has occurred or not. 0: no instruction address breakpoint has occurred on a read 1: instruction address breakpoint has occurred on a read	Read / Write	Undefined

13.2.2 Instruction Address Breakpoint Register (IAB) / Instruction Address Breakpoint Mask Register (IABM)

31 … 2	1 0
IAB	0

Figure 13-2. Instruction Address Breakpoint Register

31 … 2	1 0
IABM	0

Figure 13-3. Instruction Address Breakpoint Mask Register

This register pair holds the instruction breakpointing address. Both the value in IAB register and the current fetch PC are masked by the value in IABM. If the values are equal, condition for instruction address breakpoint becomes true. As fetch PC is always word-aligned, the bit 0 and bit 1 of these registers are fixed to zeros.

13.2.3 Data Address Breakpoint Register (DAB) / Data Address Breakpoint Mask Register (DABM)

This register pair holds the data breakpointing address. Both the value in DAB register and the destination for load/store operation are masked by the value in DABM. If the values are equal, condition for data address breakpoint becomes true. These registers are 32-bit wide readable/writable.

31 … 0
DAB

Figure 13-4. Data Address Breakpoint Register

31 … 0
DABM

Figure 13-5. Data Address Breakpoint Mask Register

13.2.4 Data Value Breakpoint Register (DVB) / Data Value Breakpoint Mask Register (DVBM)

This register pair holds the value for data value breakpointing. Both the value in DVB and the lower 32 bits of load/store data are masked with the value in DVBM. If the values are equal, condition for data value breakpoint becomes true. Note that enabling data value breakpoint implies activating the data address breakpointing (setting either/both of DRE/DWE bit in BPC), and therefore breakpoint event for data value only happens if both condition for data address breakpoint and data value breakpoint becomes true.

Note that the comparison of data value is always performed in 32bit regardless of the width of load/store operation: the store value comes from GPR is truncated to 32bit value for comparison and the load value is appropriately signextended or merged with the contents of GPR (unaligned cases) and then the least significant 32-bits are used for comparison. For instance, most significant (64+32) bits/32-bits are truncated on data value comparison for LQ/SQ/LD/SD instructions, while the value from memory is sign-extended to comprise a 32bit value for LB/LH instructions.

13.3 Setting Breakpoint

The following sections mention the details of breakpoint controls with some sample codes. As C790 is a pipelined superscalar processor, several restrictions are applied in setting breakpoint registers. The following is the main topic that has to be taken care of:

31 ... 0
DVB

Figure 13-6. Data Value Breakpoint Register

31 ... 0
DVBM

Figure 13-7. Data Value Breakpoint Mask Register

- Upon chainging the configuration of breakpointing, it is very likely that 3 or more registers must be updated. However, the change is performed in pipelined manner as C790 is pipelined processor. This potentially has possibility to create a hazardous area in generating exception unconsciously.
- C790 does NOT wait for the data arrival on load operation. The instruction itself may retire from the pipeline before storing the data into the registers, and the occurrence of breakpointing event delays from the instruction completion. This not only make some data value breakpoints imprecise, but also temporally masks an occurrence of breakpointing event as following case: a data load instruction that should cause data value breakpoint exception results in cache miss. But in the next cycle, other level2 exception such as SIO interrupt had been detected and the processor entered level2 before the acquisition of the data. Under this scenario, data value exception will be delayed until the processor returns from Level2 mode.

13.3.1 Sequence of Setting Breakpoint

In order to prevent spurious exception during reconfiguring the breakpoint, managing breakpointing enable before and after the change is mandatory. One easy way is to change the processor mode into Level2 to mask breakpoint exception unconditionally, but, this has an side effect that the user segment becomes unmapped. Therefore, this section mainly focuses on changing the configuration without changing the processor mode.
The following summarizes the sequence of changing breakpointing configuration.

1. Synchronize the pipeline
2. Disable the breakpoint exception that is going to be reconfigured
3. Synchronize the pipeline
4. Set appropriate data in Breakpoint register pairs
5. Set appropriate configuration into Breakpoint Control Register, including enabling the break point exception.
6. Synchronize the pipeline

There are three synchronization points in the sequence: the first one is to ensure that there is no pending breakpoint exception for consistency in the breakpoint exception handler. The second one is right after disabling the breakpoint that is going to be reconfigured. This separates the change in the control register from the change for other breakpoint register so that programmer can safely change the breakpoint. The third synchronization is after updating breakpoint control register. Since C790 issues the instructions in in-ordered manner, changes for breakpoint register pair always precedes the change in the control register. In this sense, there is no spurious exception without this synchronization. However, in order to catch the breakpointing event right after updating the control register, flushing the pipeline at this point is strongly recommended.
The first synchronized operation must be either of SYNC.P or SYNC.L operation depending on the breakpoint that is going to be reconfigured. If it is instruction breakpoint, SYNC.P is to be used and otherwise SYNC.L is to be used. For second and third synchronization, SYNC.P is to be used.
The flow generating TRIG* and exception is shown in Figure 13-8, Figure 13-9, Figure 13-10. Figure 13-8 describes the flow hardware breakpoint encounts the breakpointing event. Figure 13-9, and Figure 13-10 describe the flow how the exception and TRIG* signal is asserted.
The following shows some simple sample codes for configuring breakpoint registers. Several programming notes/issues are put in the comments.

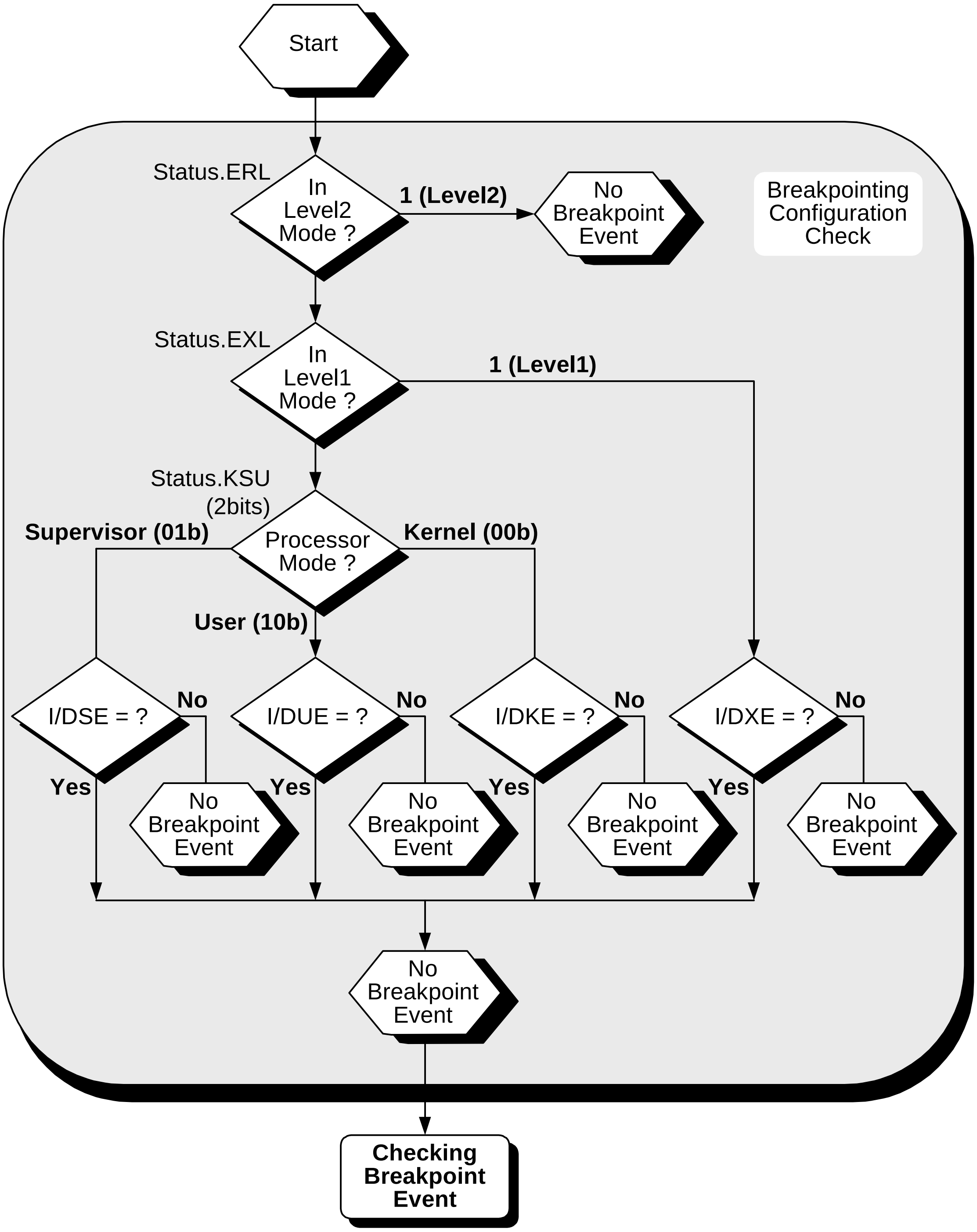


Figure 13-8. Hardware Breakpoint detection flow (Setting)

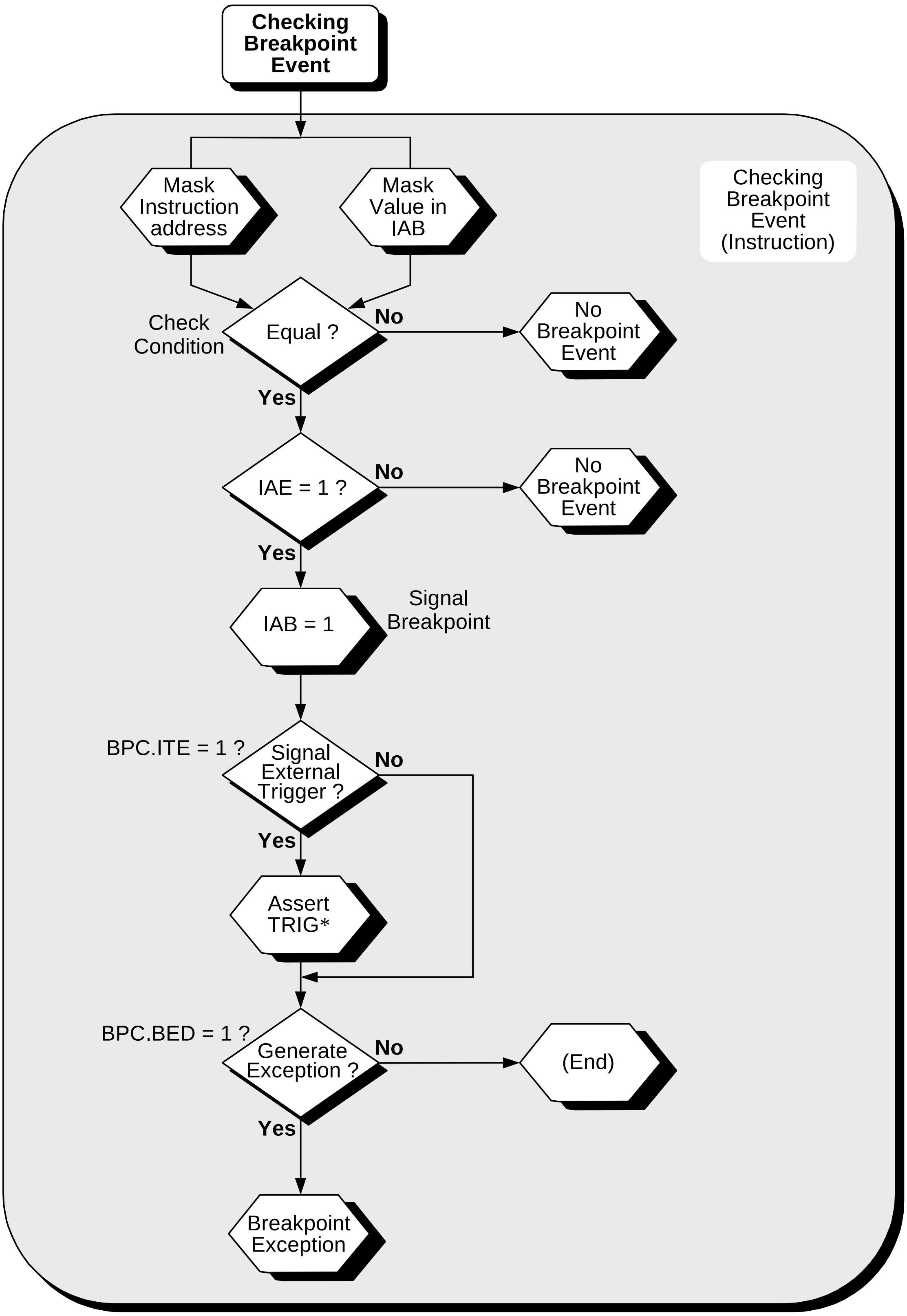


Figure 13-9. Hardware Breakpoint detection flow (IAB)

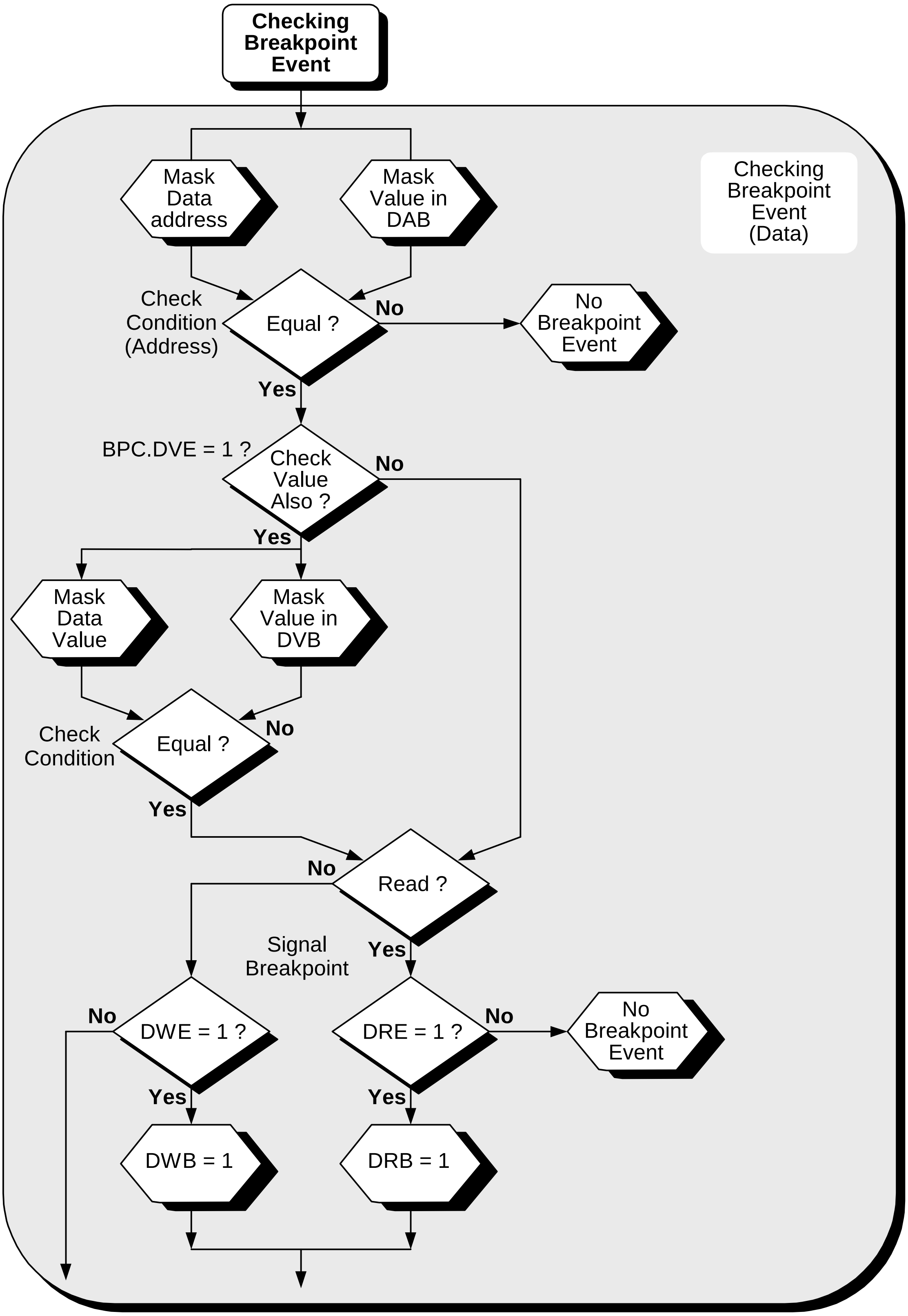


Figure 13-10. Hardware Breakpoint detection flow (DAB/DVB) (1/2)

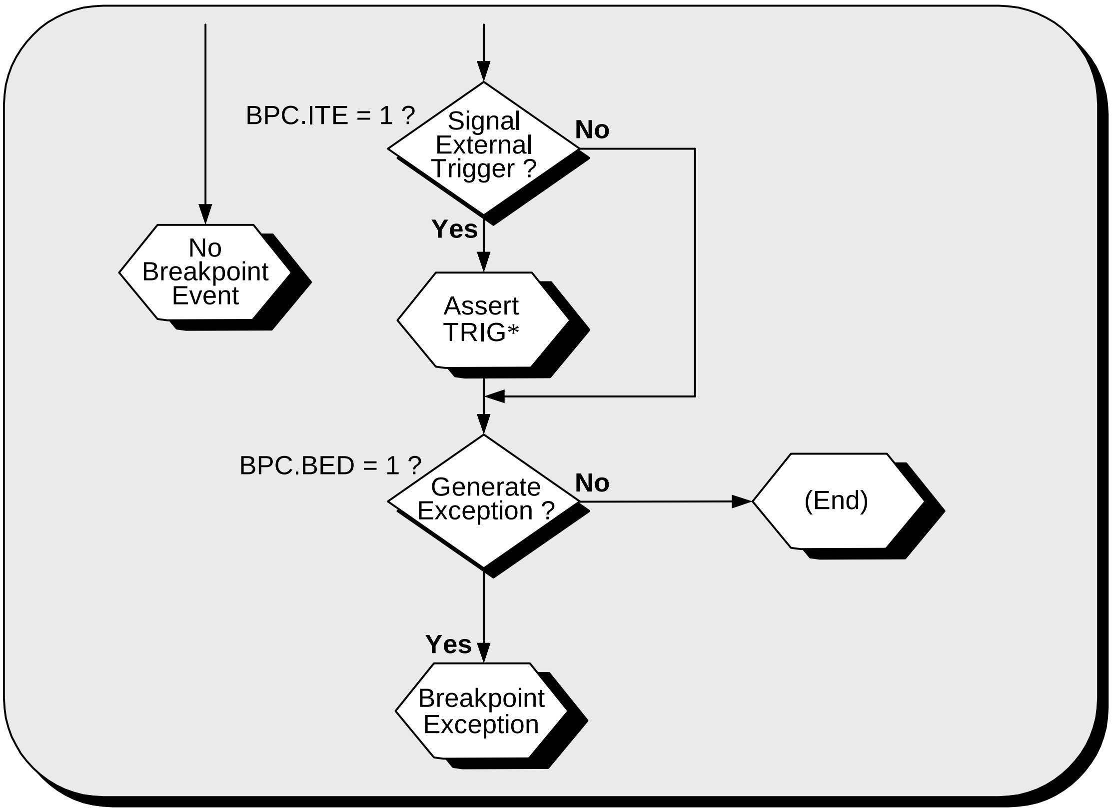


Figure 13-10. Hardware Breakpoint detection flow (IAB) (2/2)

13.3.2 Instruction Breakpointing

The following code sets an instruction breakpoint from 0x1234_5600 to 0x1234_56ff, and traps if the processor is either in user mode or in supervisor mode.

```
-----------------------------------------------------------------
#
# Setting Instruction address breakpoint from 0x1234_5600 to 0x1234_56ff
# in user mode and supervisor mode
#
# 1st sync.
sync.p                   # A barrier to ensure there is no pending
                         # instruction address breakpoint in pipe.
                         # pipeline flusing works for this purpose.

# At first, disable instruction breakpointing to avoid spurious exceptions.
# The following uses conservative way not to break the configuration for
# data breakpointing.
#
mfbpc  $4                # get the value in BPC
bgez   $4, 1f            # skip following if ( BPC[31] == 0 )
nop                      # (bds)
li     $5, (1 << 31)     # IAE is in 31st bit of BPC
xor    $4, $5, $4        # Resetting IAE bit to zero.
mtbpc  $4                # reload BPC.

# 2nd sync.
sync.p                   # barrier to ensure the configuration change
                         # of breakpoint function
1:
#
# Reconfigure instruction breakpoint address.
# Note that least significant 8 bits can be anything because it is masked
# by IABM register anyway
#
li     $4, 0x12345678
mtiab  $4

#
# Setting mask register. Masked if corresponding bit in mask register
# is reset to zero.
#
li     $5, 0xffffff00
mtiabm $5

#
# Reconfigure instruction breakpoint. For better understanding, once
# resetting all the bits for instructio breakpoint, and then sets new
# config.
#
mfbpc  $4

#
# Reset IUE/ISE/IKE/ITE/IAB. Especially resetting IAB is important to
# know the cause of next breakpoint exception correctly.
#
li     $5, ~(                         \
                ( 1 << 26 )    # IUE  \
              | ( 1 << 25 )    # ISE  \
              | ( 1 << 24 )    # IKE  \
              | ( 1 << 23 )    # IXE  \
              | ( 1 << 17 )    # ITE  \
              | ( 1 <<  0 )    # IAB  \
            )
and    $4, $4, $5

#
# Set new configuration to BPC register.
# Note that setting BPC after IAB/IABM is so important to avoid spurious
# exception.
#
```

```
li      $6, $6,                                                       \
             (                                                        \
                ( 1 << 31 ) # IAE = 1 to enable Inst. B.P.            \
              | ( 1 << 26 ) # IUE = 1 to enable Inst. B.P in user mode.  \
              | ( 1 << 20 ) # IUE = 1 to enable Inst. B.P in supv. mode. \
              | ( 1 << 15 ) # BED = 1 to enable generating exception.    \
                )
or      $5, $4, $6
mtbpc   $5

# 3rd sync.
Sync.p                      # Barrier to ensure the configuration change
-----------------------------------------------------------------
```

13.3.3 Data Address Breakpointing

The following code sets a data address breakpoint from 0x1230_0000 to 0x1233_ffff for both reading and writing, and traps if the processor is either in kernel mode(including under level1).

```
-----------------------------------------------------------------
#
# Setting data address breakpoint from 0x1230_0000 to 0x1233_ffff
# in kernel(normal,L1) mode
#
# 1st sync.
sync.l                  # A barrier to ensure there is no pending
                        # data address breakpoint in pipe.
                        # Must flush all buffers for load/store for this
                        # purpose by SYNC.L
#
# At first, reset data-breakpoint related bits to zeros.
# Resetting DWB/DRB is important so that the hander can recognize the
# next breakpoint exception correctly.
#
mfbpc  $4               # load current configuration
li     $5, ~(                                \
                ( 1 << 30 )    # DRE  \
              | ( 1 << 29 )    # DWE  \
              | ( 1 << 28 )    # DVE  \
              | ( 1 << 21 )    # DUE  \
              | ( 1 << 20 )    # DSE  \
              | ( 1 << 19 )    # DKE  \
              | ( 1 << 18 )    # DXE  \
              | ( 1 << 16 )    # DTE  \
              | ( 1 << 2  )    # DWB  \
              | ( 1 << 1  )    # DRB  \
            )
and    $4, $4, $5
mtbpc  $4               # reload BPC.

# 2nd sync.
sync.p                  # barrier to ensure the configuration change
                        # of breakpoint function
#
# Reconfigure data breakpoint address.
# Note that least significant 18 bits can be anything because it is masked
# by DABM register anyway
#
li     $6, 0x12305678
mtdab  $6

#
# Setting mask register. Masked if corresponding bit in mask register
# is reset to zero.
#
li     $5, 0xfffc0000
mtdabm $5

#
# Set new configuration to BPC register.
# Note that setting BPC after DAB/DABM is so important to avoid spurious
# exception.
#
li     $6, $6,                                                    \
            (                                                     \
                ( 1 << 30 ) # DRE = 1 to enable Data B.P on read         \
              | ( 1 << 29 ) # DWE = 1 to enable Data B.P on write        \
              | ( 1 << 19 ) # DKE = 1 to enable Data B.P in kern. mode.  \
              | ( 1 << 18 ) # DXE = 1 to enable Data B.P under L1.       \
              | ( 1 << 15 ) # BED = 1 to enable generating exception.    \
                )
or     $5, $4, $6          # Note that $4 still holds the value used
                           # on MTBPC.
mtbpc  $5
```

```
# 3rd sync.
sync.p                          # Barrier to ensure the configuration change
------------------------------------------------------------------
```

13.3.4 Breakpointing by Data Address and Value

Setting Data Address and Value breakpoint is the same as Data Address breakpoint. The following example is the same as the previous example except in that the trap only happens if the data contains 0xCAFE in least significant 16 bits, and traps only on loading data.

```
-----------------------------------------------------------------
#
# Setting data address/value breakpoint from 0x1230_0000 to 0x1233_ffff
# with data that contains 0xCAFE in kernel(normal, L1) mode.
#
# 1st sync.
sync.l                   # A barrier to ensure there is no pending
                         # data address breakpoint in pipe.
                         # Must flush all buffers for load/store for this
                         # purpose by SYNC.L
#
# At first, reset data-breakpoint related bits to zeros.
# Resetting DWB/DRB is important so that the hander can recognize the
# next breakpoint exception correctly.
#
mfbpc  $4                # load current configuration
li     $5, ~(                        \
               ( 1 << 30 )    # DRE  \
             | ( 1 << 29 )    # DWE  \
             | ( 1 << 28 )    # DVE  \
             | ( 1 << 21 )    # DUE  \
             | ( 1 << 20 )    # DSE  \
             | ( 1 << 19 )    # DKE  \
             | ( 1 << 18 )    # DXE  \
             | ( 1 << 16 )    # DTE  \
             | ( 1 << 2  )    # DWB  \
             | ( 1 << 1  )    # DRB  \
            )
and    $4, $4, $5
mtbpc  $4                # reload BPC.

# 2nd sync.
sync.p                   # barrier to ensure the configuration change
                         # of breakpoint function
#
# Reconfigure data breakpoint address.
# Note that least significant 18 bits can be anything because it is masked
# by DABM register anyway
#
li     $6, 0x1233ffff
mtdab  $6

#
# Setting mask register. Masked if corresponding bit in mask register
# is reset to zero.
#
li     $5, 0xfffc0000
mtdabm $5

#
# Configure data value address.
# Note that least significant 8 bits can be anything because it is masked
# by DVBM register anyway
#
li     $6, 0xbabecafe
mtdvb  $6

#
# Setting mask register. Masked if corresponding bit in mask register
# is reset to zero.
#
li     $5, 0x0000ffff
mtdvbm $5
```

```
#
# Set new configuration to BPC register.
# Note that setting BPC after DAB/DABM is so important to avoid spurious
# exception.
#
li     $6,                                                              \
             (                                                          \
                ( 1 << 30 ) # DRE = 1 to enable Data B.P on read         \
              | ( 1 << 28 ) # DVE = 1 to enable Data value B.P           \
              | ( 1 << 19 ) # DKE = 1 to enable Data B.P in kern. mode.  \
              | ( 1 << 18 ) # DXE = 1 to enable Data B.P under L1.       \
              | ( 1 << 15 ) # BED = 1 to enable generating exception.    \
                )
or     $5, $4, $6           # Note that $4 still holds the value used
                            # on MTBPC.
mtbpc  $5

# 3rd sync.
sync.p                      # Barrier to ensure the configuration change
------------------------------------------------------------------
```

13.3.5 Data Value Breakpointing

Data value breakpoint can be configured so that it traps only by data value, by setting zero to *DABM* register and configuring the data breakpoint to “Data Address and Value” mode.

13.4 Triggering External Probes

There is one dedicated pad to make breakpoint visible outside of C790. This pad, TRIG* signal, is asserted for two cycles whenever break point event is detected. This trigger signal generation is enabled by setting ITE/DTE bit in *BPC* register to 1. Note that assertion of TRIG* signal is not completely synchronized with the occurrence of exception: TRIG signal is directly connected to the internal breakpoint detect logic while exception including breakpoint always occurs along with retirement of instruction. Threfore, thiming of the assertion of TRIG* signal and that of occurrence of exception may differs. Especially, if the breakpoint is detected right before entering Level2 mode, and if the breakpoint exception is taken imprecisely, exception may be masked because of processor's mode change although TRIG* signal has already been asserted.

13.5 Important notice on using hardware breakpoint

One important issue not mentioned in this section is that breakpointing does not take care of ASID on detecting breakpoint. This implies not only that software has to take care of it on context switching to apply breakpointing for a specific process, but also that imprecise breakpoint exception may be detected after or in the middle of context switching. In such condition, it may become difficult to identify which process the breakpoint exception belongs to. This can be avoided by executing SYNC.L instruction right before changing ASID. (Since all imprecise breakpoint events relates to load/store instructions, executing SYNC.L works as a barrier)

Relating to this issue, as briefly described in section 13.3, issuing breakpoint exception may delay because of other level2 exception handling, although the breakpoint exception is actual precedent from instruction ordering point of view. In such condition, because C790 generates breakpoint exception after the processor returns from Level2,[1] there is no possibility to miss encounting the breakpoint. However, if the program need to insure the order of occurrence between level2 exceptions, software has to take care of it (i.e. all level2 handler has to check the occurrence of breakpointing first). Similarly, if a level2 exception DOES NOT return to where the exception was detected, software has to insure to reset the condition of breakpoint.

[1] C790 tracks the occurrence of breakpoint exception until the breakpoint exception is taken.

INDEX

A. CPU Instruction Set Details

This appendix provides a detailed description of the operation of each instruction. The instructions are listed in alphabetical order.

Exceptions that may occur due to the execution of each instruction are listed after the description of each instruction. Descriptions of the immediate cause and manner of handling exceptions are omitted from the instruction descriptions in this appendix.

Descriptions use a pseudocode notation explained in Section A.2.

For an overview of the instruction set, refer to Chapter 3 of the User's Manual.

A.1 Description of an Instruction

Each instruction description contains several sections that contain specific information about the instruction. The following sections describe the contents of each section in detail.

A.1.1 Instruction Mnemonic and Name

The instruction mnemonic and name are printed as page headings for each page in the instruction description.

A.1.2 Instruction Encoding Picture

The instruction word encoding is shown in pictorial form at the top of the instruction description. The picture shows the values of all constant fields and the opcode names for opcode fields in upper-case. It labels all variable fields with lower-case names that are used in the instruction description. Fields that contain zeroes but are not named are unused fields that are required to be zero.

A.1.3 Format

The assembler formats for the instruction and the architecture level at which the instruction was originally defined are shown.

A.1.4 Purpose

This is a very short statement of the purpose of the instruction.

A.1.5 Description

If a one-line symbolic description of the instruction is feasible, it will appear immediately to the right of the *Description* heading. The body of the section is a description of the operation of the instruction in text, tables, and figures. This description complements the high-level language description in the *Operation* section.

A.1.6 Restrictions

This section documents the restrictions on the instructions. Most restrictions fall in the category of alignment requirements for memory addresses, valid values of operands, and order of instructions necessary to gurantee correct execution.

A.1.7 Operation

This section describes the operation as pseudocode in a high-level language notation resembling Pascal. The purpose of this section is to describe the operation of the instruction clearly in a form with less ambiguity than prose.

A.1.8 Exceptions

This section lists the exceptions that can be caused by the **operation** of the instruction. It omits exceptions that can be caused by instruction fetch, performance counters, and breakpoints. It also omits exceptions that can be caused by asynchronous external events, e.g. interrupts. Although the Bus Error exception may be caused by the operation of a load, store or PREF instruction this section does not list Bus Error for load, store or PREF instructions because the relationship between these instructions and external error conditions, like Bus Error is asynchronous and implementation specific.

A.1.9 Programming Notes, Implementation Notes

These sections contain material that is useful for programmers and implementors respectively but is not necessary to describe the instruction and does not belong in the description sections.

A.2 Instruction Description Notation and Functions

The *Operation* sections of the instruction descriptions describe the operation performed by each instruction using a high-level language notation, or pseudocode. Symbols, functions, and structures used in the *Operation* sections are described here.

A.2.1.1 Pseudocode Language Statement Execution

Each of the high-level language statements in an operation description is executed in sequential order (as modified by conditional and loop constructs).

A.2.1.2 Pseudocode Symbols

Special symbols used in the notation are described in Table A-1.

Table A-1. Symbols in Instruction Operation Statements

Symbol	Meaning
←	Assignment.
=, ≠	Tests for equality and inequality.
||	Bit string concatenation.
X^y	A *y*-bit string formed by *y* copies of the single-bit value *x*.
Xy..z	Selection of bits *y* through *z* of bit string *x*.
+, –	Two's complement or floating point arithmetic: addition, subtraction.
*, ×	Two's complement or floating point multiplication (both used for either).
div	Two's complement integer division.
Mod	Two's complement modulo.
/	Floating point division.
<	Two's complement less than comparison.
Not	Bit-wise logical NOT.
Nor	Bit-wise logical NOR.
Xor	Bit-wise logical XOR.
And	Bit-wise logical AND.
or	Bit-wise logical OR.
GPRLEN	The length in bits (64 in the C790), of the CPU General Purpose Registers.
GPR[x]	CPU General Purpose Register *x*. The content of *GPR[0]* is always zero.
CPR[z, x]	Coprocessor unit *z*, general register *x*.
CCR[z, x]	Coprocessor unit *z*, control register *x*.
CPCOND[z]	Coprocessor unit *z* condition signal.
BigEndian	Big-endian made as configured at reset (0→Little, 1→Big) from core boundary signal.

<table>
<tr><th>Symbol</th><th>Meaning</th></tr>
<tr><td>I:,
I+n:,
I–n:</td><td>This occurs as a prefix to operation description lines and functions as a label. It indicates the instruction time during which the effects of the pseudocode lines appears to occur (i.e., when the pseudocode is “executed”). Unless otherwise indicated, all effects of the current instruction appear to occur during the instruction time of the current instruction.
No label is equivalent to a time label of “I:”.
Sometimes effects of an instruction appear to occur either earlier or later-during the instruction time of another instruction. When that happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction I, in which the effect of that pseudocode appears to occur. For example, an instruction may have a result that is not available until after the next instruction. Such an instruction will have the portion of the instruction operation description that writes the result register in a section labeled “I+1:”.
The effect of pseudocode statements for the current instruction labeled “I+1:” appears to occur “at the same time” as the effect of pseudocode statements labeled “I:” for the following instruction. Within one pseudocode sequence the effects of the statements takes place in order. However, between sequences of statements for different instructions that occur “at the same time”, there is no order defined. Programs must not depend on a particular order of evaluation between such sections.</td></tr>
<tr><td>PC</td><td>The Program Counter value. During the instruction time of an instruction this is the address of the instruction word. The address of the instruction that occurs during the next instruction time is determined by assigning a value to PC during an instruction time. If no value is assigned to PC during instruction time by any pseudocode statement, it is automatically incremented by 4 before the next instruction time. A taken branch assigns the target address to PC during the instruction time of the instruction in the branch delay slot.</td></tr>
<tr><td>PSIZE</td><td>The SIZE, number of bits, of Physical address in an implementation.</td></tr>
</table>

A.2.2 Definitions of Pseudocode Functions Used in Instruction Descriptions

A variety of functions are used in the pseudocode employed in the instruction descriptions. These functions are used to make the pseudocode more readable and also to abstract implementation-specific behavior. These functions are defined in this section. Certain additional functions specific to a particular coprocessor are described at the beginning of the appendix for that coprocessor.

A.2.2.1 Coprocessor General Register Access Pseudocode Functions

Defined coprocessors, except for COP0, have instructions to exchange words and doublewords and quadwords between coprocessor general registers and the rest of the system. What a coprocessor does with a word or doubleword supplied to it, and how a coprocessor supplies a word or doubleword, is defined by the coprocessor itself. The functions are listed in Table A-2.

Table A-2. Coprocessor General Register Access Functions

COP_LW(z, rt, memword)		
	z:	The coprocessor unit number.
	rt:	Coprocessor general register specifier.
	Memword:	A 32-bit word value supplied to the coprocessor.
This is the action taken by coprocessor z when supplied with a word from memory during a load word operation. The action is coprocessor-specific. The typical action would be to store the contents of memword in coprocessor general register *rt*.		
COP_LD(z, rt, memdouble)		
	z:	The coprocessor unit number.
	rt:	Coprocessor general register specifier.
	Memdouble:	64-bit doubleword value supplied to the coprocessor.
This is the action taken by coprocessor z when supplied with a doubleword from memory during a load doubleword operation. The action is coprocessor-specific. The typical action would be to store the contents of memdouble in coprocessor general register *rt*.		
Dataword ← COP_SW(z, rt)		
	z:	The coprocessor unit number.
	rt:	Coprocessor general register specifier.
	Dataword:	32-bit word value.
This defines the action taken by coprocessor z to supply a word of data during a store word operation. The action is coprocessor-specific. The typical action would be to supply the contents of low-order word in coprocessor general register *rt*.		
Datadouble ← COP_SD(z, rt)		
	z:	The coprocessor unit number.
	rt:	Coprocessor general register specifier.
	Datadouble:	64-bit doubleword value.
This defines the action taken by coprocessor z to supply a doubleword of data during a store doubleword operation. The action is coprocessor-specific. The typical action would be to supply the contents of the doubleword coprocessor general register *rt*.		

A.2.2.2 Load and Store Memory Pseudocode Functions

Regardless of byte-numbering order (endianness), the address of a halfword, word, or doubleword is the smallest byte address among the bytes in the object. For a big-endian ordering this is the most-significant byte; for a little-endian ordering this is the least-significant byte.

In the operation description pseudocode for load and store operations, the functions listed in Table A-3 are used to summarize the handling of virtual addresses and accessing physical memory.

The size of the data item to be loaded or stored is passed in the *AccessLength* field. The valid constant names and values are shown in Table A-4. The bytes within the addressed unit of memory (quadword for 128-bit processors) which are used can be determined directly from the AccessLength and the four low-order bits of the address.

Table A-3. Load and Store Functions

(pAddr, CCA) ← AddressTranslation (vAddr, IorD, LorS)

pAddr:	Physical Address.
CCA:	Cache Coherence Algorithm: the method used to access caches and memory and resolve the reference.
vAddr:	Virtual Address.
IorD:	Indicates whether access is for Instruction or Data.
LorS:	Indicates whether access is for Load or Store

Translate a virtual address to a physical address and a cache coherence algorithm describing the mechanism used to resolve the memory reference.

Given the virtual address vAddr, and whether the reference is to Instructions or Data (IorD), find the corresponding physical address (pAddr) and the cache coherence algorithm (CCA) used to resolve the reference. If the virtual address is in one of the unmapped address spaces the physical address and CCA are determined directly by the virtual address. If the virtual address is in one of the mapped address spaces then the TLB is used to determine the physical address and access type; if the required translation is not present in the TLB or the desired access is not permitted the function fails and an exception is taken.

MemElem ← LoadMemory (CCA, AccessLength, pAddr, vAddr, IorD)

MemElem:	Data is returned in a fixed width with a natural alignment. The width is the same size as the CPU general purpose register.
CCA:	Cache Coherence Algorithm: the method used to access caches and memory and resolve the reference.
AccessLength:	Length, in bytes, of access.
pAddr:	Physical Address.
vAddr:	Virtual Address.
IorD:	Indicates whether access is for Instructions or Data.

Load a value from memory.

Uses the cache and main memory as specified in the Cache Coherence Algorithm (CCA) and the sort of access (IorD) to find the contents of AccessLength memory bytes starting at physical location pAddr. The data is returned in the fixed width naturally-aligned memory element (MemElem). The low-order two, three, or four bits of the address and the AccessLength indicate which of the bytes within MemElem needs to be given to the processor. If the memory access type of the reference is uncached then only the referenced bytes are read from memory ad valid within the memory element. If the access type is cached, and the data is not present in cache, an implementation specific size and alignment block of memory is read and loaded into the cache to satisfy a load reference. At a minimum, the block is the entire memory element.

StoreMemory (CCA, AccessLength, MemElem, pAddr, vAddr)

CCA: Cache Coherence Algorithm: the method used to access caches and memory and resolve the reference.

AccessLength: Length, in bytes, of access.

MemElem: Data in the width and alignment of a memory element. The width is the same size as the CPU general purpose register. For a partial-memory-element store, only the bytes that will be stored must be valid.

pAddr: Physical Address.

vAddr: Virtual Address.

Store a value to memory.

The specified data is stored into the physical location pAddr using the memory hierarchy (data caches and main memory) as specified by the Cache Coherence Algorithm (CCA). The MemElem contains the data for an aligned, fixed-width memory element, though only the bytes that will actually be stored to memory need to be valid. The low-order four bits of pAddr and the AccessLength field indicates which of the bytes within the MemElem data should actually be stored; only these bytes in memory will be changed.

Prefetch (CCA, pAddr, vAddr, DATA, hint)

CCA: Cache Coherence Algorithm: the method used to access caches and memory and resolve the reference.

pAddr: Physical Address.

vAddr: Virtual Address.

DATA: Indicates that access is for DATA.

hint: Hint that indicates the possible use of the data

Prefetch data from memory.

Prefetch is an advisory instruction for which an implementation specific action is taken. The action taken may increase performance but must not change the meaning of the program or alter architecturally-visible state.

Table A-4. AccessLength Specifications for Loads / Stores

AccessLength name	Value	Meaning
QUADWORD	15	16 bytes (128 bits)
DOUBLEWORD	7	8 bytes (64 bits)
SEPTIBYTE	6	7 bytes (56 bits)
SEXTIBYTE	5	6 bytes (48 bits)
QUINTIBYTE	4	5 bytes (40 bits)
WORD	3	4 bytes (32 bits)
TRIPLEBYTE	2	3 bytes (24 bits)
HALFWORD	1	2 bytes (16 bits)
BYTE	0	1 byte (8 bits)

A.2.2.3 Miscellaneous Functions

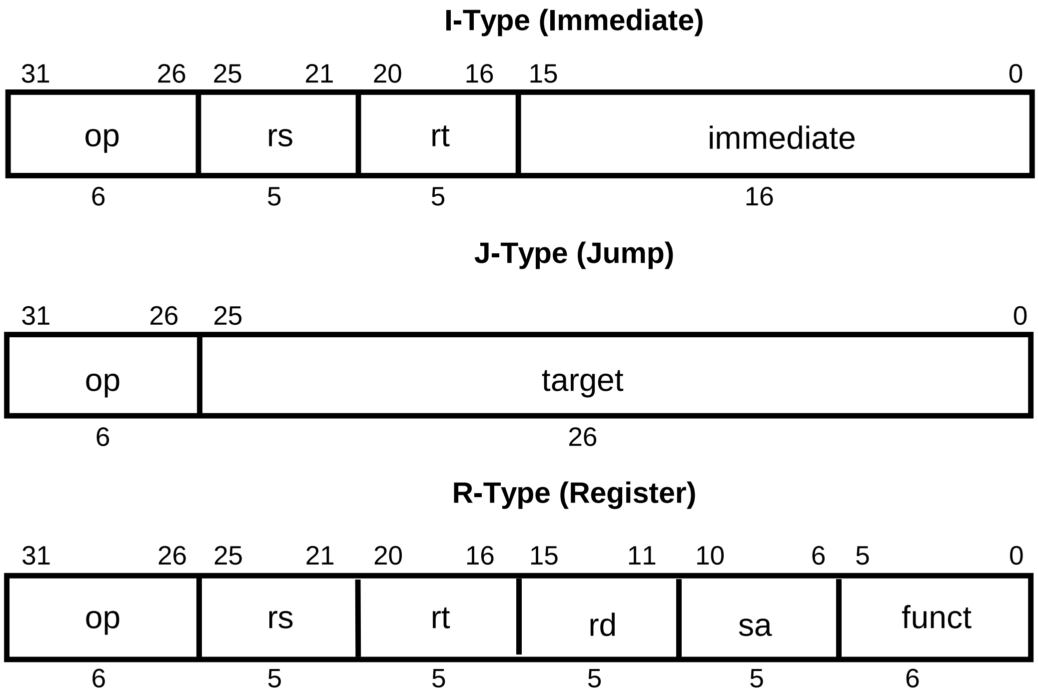
Table A-5 describes additional miscellaneous functions for CPU instruction descriptions.

Table A-5. Miscellaneous Functions

SyncOperation (stype) stype: Type of synchronization operation to be performed. Based on the value of stype either a memory barrier operation is performed or a pipeline barrier operation is performed. In case of a memory barrier all pending loads and stores are retired. Loads are retired when the destination register is written. Stores are retired when the stored data (in store buffers or write buffers) is either stored in the data cache, or sent on the processor bus. All uncached accelerated data gathering operation is terminated. The uncached accelerated buffer is invalidated. All bus read processes due to load/store/pref/cache instructions are completed. All pending bus write processes in the write back buffer are completed. In case of pipeline barrier all instructions prior to the barrier are completed before the instructions following the barrier operation are fetched. Note that the barrier operation does not wait for any instruction which was issued prior to the barrier operation but not retired (e.g., multiply, divide, multicycle COP1 operations or a pending load which were issued prior to the pipeline barrier operation).
SignalException (Exception) Exception; The exception condition that exists. Signal an exception condition. This will result in an exception that aborts the instruction. The instruction operation pseudocode will never see a return from this function call.
UndefinedResult() This function indicates that the result of the operation is undefined.
NullifyCurrentInstruction() Nullify the current instruction. This occurs during the instruction time for some instruction and that instruction is not executed further. This appears for branch-likely instructions during the execution of the instruction in the delay slot and it kills the instruction in the delay slot.
CoprocessorOperation (z, cop_fun) z: Coprocessor unit number cop_fun: Coprocessor function from function field of instruction Perform the specified Coprocessor operation.

A.3 CPU Instruction Formats

A CPU instruction is a single 32-bit aligned word. There are three instruction formats: Immediate (I-type), Jump (J-type), and Register (R-type). These formats are shown in Figure A-1 below:



op	6-bit primary operation code
rd	5-bit destination register specifier
rs	5-bit source register specifier
rt	5-bit target (source/destination) register specification or branch condition
immediate	16-bit signed immediate used for: logical operands, arithmetic signed operands, load/store address byte offsets, PC-relative branch signed instruction displacement
target	26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address.
sa	5-bit shift amount
funct	6-bit function field used to specify functions within the primary operation code value SPECIAL

Figure A-1. CPU Instruction Formats

A.4 Instruction Descriptions

The user-level CPU instructions are described in alphabetical order in this section.

ADD Add Word ADD

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000
6	5	5	5	5	6

MIPS I

Format: ADD rd, rs, rt

Purpose: To add 32-bit integers. If overflow occurs, then trap.

Description: rd ← rs + rt

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result. If the addition results in 32-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

```
If (NotWordValue (GPR[rs] 63..0) or NotWordValue (GPR[rt] 63..0)) then UndefinedResult()endif
temp ← GPR[rs] 63..0 + GPR[rt] 63..0
if (32_bit_arithmetic_overflow) then
      SignalException (IntegerOverflow)
else
      GPR[rd]63..0 ← sign_extend (temp31..0)
endif
```

Exceptions:

Integer Overflow

Programming Notes:

ADDU performs the same arithmetic operation but, does not trap on overflow.

ADDI Add Immediate Word ADDI

31 ... 26	25 ... 21	20 ... 16	15 ... 0
ADDI 001000	rs	rt	immediate
6	5	5	16

MIPS I

Format: ADDI rt, rs, immediate

Purpose: To add a constant to a 32-bit integer. If overflow occurs, then trap.

Description: rt ← rs + immediate

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* to produce a 32-bit result. If the addition results in 32-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rt*.

Restrictions:

If GPR *rs* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is undefined.

Operation:

```
if (NotWordValue (GPR[rs] 63..0)) then UndefinedResult() endif
temp ← GPR[rs] 63..0 + sign_extend (immediate)
if (32_bit_arithmetic_overflow) then
        SignalException (IntegerOverflow)
else
        GPR[rt]63..0 ← sign_extend (temp31..0)
endif
```

Exceptions:

Integer Overflow

Programming Notes:

ADDIU performs the same arithmetic operation but, does not trap on overflow.

ADDIU — Add Immediate Unsigned Word — ADDIU

31–26	25–21	20–16	15–0
ADDIU 001001	rs	rt	immediate
6	5	5	16

MIPS I

Format: ADDIU rt, rs, immediate

Purpose: To add a constant to a 32-bit integer.

Description: rt ← rs + immediate

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rt*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

If GPR *rs* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is undefined.

Operation:

if (NotWordValue (GPR[rs] $_{63..0}$)) then UndefinedResult( ) endif
temp ← GPR[rs] $_{63..0}$ + sign_extend (immediate)
GPR[rt] $_{63..0}$ ← sign_extend (temp$_{31..0}$)

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

ADDU — Add Unsigned Word — ADDU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADDU 100001
6	5	5	5	5	6

MIPS I

Format: ADDU rd, rs, rt

Purpose: To add 32-bit integers.

Description: rd ← rs + rt

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rd.*

No Integer Overflow exception occurs under any circumstances.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

if (NotWordValue (GPR[rs] $_{63..0}$) or NotWordValue (GPR[rt] $_{63..0}$)) then UndefinedResult() endif
temp ← GPR[rs] $_{63..0}$ + GPR[rt] $_{63..0}$
GPR[rt] $_{63..0}$ ←sign_extend (temp$_{31..0}$)

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

AND

And

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	AND 100100
6	5	5	5	5	6

MIPS I

Format: AND rd, rs, rt

Purpose: To do a bitwise logical AND.

Description: rd ← rs AND rt

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical AND operation. The result is placed into GPR *rd*.

Restrictions:

None

Operation:

$GPR[rd]_{63..0} \leftarrow GPR[rs]_{63..0}$ and $GPR[rt]_{63..0}$

Exceptions:

None

Programming Notes:

None

ANDI

And Immediate

31–26	25–21	20–16	15–0
ANDI 001100	rs	rt	immediate
6	5	5	16

MIPS I

Format: ANDI rt, rs, immediate

Purpose: To do a bitwise logical AND with a constant.

Description: rt ← rs AND immediate

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR *rs* in a bitwise logical AND operation. The result is placed into GPR *rt*.

Restrictions:

None

Operation:

$GPR[rt]_{63..0} \leftarrow$ zero_extend (immediate) and $GPR[rs]_{63..0}$

Exceptions:

None

Programming Notes:

None

BEQ — Branch on Equal — BEQ

31–26	25–21	20–16	15–0
BEQ 000100	rs	rt	offset
6	5	5	16

MIPS I

Format: BEQ rs, rt, offset

Purpose: To compare GPRs then do a PC-relative conditional branch.

Description: if (rs = rt) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are equal, branch to the effective target address after the instruction in the delay slot is executed.

Restriction:

None

Operation:

```
I:    tgt_offset ← sign_extend (offset || 0²)
      condition ← (GPR[rs] 63..0 = GPR[rt] 63..0)
I+1:  if condition then
              PC ← PC + tgt_offset
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BEQL Branch on Equal Likely BEQL

31 26	25 21	20 16	15 0
BEQL 010100	rs	rt	offset
6	5	5	16

MIPS II

Format: BEQL rs, rt, offset

Purpose: To compare GPRs then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (rs = rt) then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are equal, branch to the target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend (offset || 0^2)
      condition ← (GPR[rs] 63..0 = GPR[rt] 63..0)
I+1:  if condition then
              PC ← PC + tgt_offset
      else
              NullifyCurrentInstruction()
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BGEZ — Branch on Greater Than or Equal to Zero — BGEZ

31–26	25–21	20–16	15–0
REGIMM 000001	rs	BGEZ 00001	offset
6	5	5	16

MIPS I

Format: BGEZ rs, offset

Purpose: To test a GPR then do a PC-relative conditional branch.

Description: if (rs ≥ 0) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend (offset || 0^2)
      condition  ← GPR[rs] 63..0 ≥ 0^GPRLEN
I+1:  if condition then
          PC ← PC + tgt_offset
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BGEZAL Branch on Greater Than or Equal to Zero and Link BGEZAL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BGEZAL 10001	offset
6	5	5	16

MIPS I

Format: BGEZAL rs, offset

Purpose: To test a GPR then do a PC-relative conditional procedure call.

Description: if (rs ≥ 0) then procedure_call

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution would continue after a procedure call.

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed.

Restriction:

GPR 31 must not be used for the source register *rs*, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

Operation:

I: tgt_offset ← sign_extend (offset || 0^2)
condition ← $GPR[rs]_{63..0} \geq 0^{GPRLEN}$
$GPR[31]_{63..0}$ ← zero_extend (PC+8)
I+1: if condition then
PC ← PC + tgt_offset
endif

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to more distant addresses.

BGEZALL Branch on Greater Than or Equal to Zero and Link Likely BGEZALL

31–26	25–21	20–16	15–0
REGIMM 000001	rs	BGEZALL 10011	offset
6	5	5	16

MIPS II

Format: BGEZALL rs, offset

Purpose: To test a GPR then do a PC-relative conditional procedure call; execute the delay slot only if the branch is taken.

Description: if (rs ≥ 0) then procedure_call_likely

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution would continue after a procedure call.

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

GPR 31 must not be used for the source register *rs*, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

Operation:

```
I:    tgt_offset ← sign_extend (offset || 0^2)
      condition ← GPR[rs]63..0 ≥ 0^GPRLEN
      GPR[31]63..0 ← zero_extend (PC+8)
I+1:  if condition then
              PC ← PC + tgt_offset
      else
              NullifyCurrentInstruction()
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to more distant addresses.

BGEZL Branch on Greater Than or Equal to Zero Likely BGEZL

31–26	25–21	20–16	15–0
REGIMM 000001	rs	BGEZL 00011	offset
6	5	5	16

MIPS II

Format: BGEZL rs, offset

Purpose: To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (rs ≥ 0) then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend (offset || 0^2)
      condition ← GPR[rs]63..0 ≥ 0^GPRLEN
I+1:  if condition then
              PC ← PC + tgt_offset
      else
              NullifyCurrentInstruction()
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BGTZ Branch on Greater Than Zero BGTZ

31 26	25 21	20 16	15 0
BGTZ 000111	rs	0 00000	offset
6	5	5	16

MIPS I

Format: BGTZ rs, offset

Purpose: To test a GPR then do a PC-relative conditional branch.

Description: if (rs > 0) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than zero (sign bit is 0 but value not zero), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend (offset || 0^2)
      condition ← GPR[rs]_63..0 > 0^GPRLEN
I+1:  if condition then
              PC ← PC + tgt_offset
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BGTZL

Branch on Greater Than Zero Likely

31 … 26	25 … 21	20 … 16	15 … 0
BGTZL 010111	rs	0 00000	offset
6	5	5	16

MIPS II

Format: BGTZL rs, offset

Purpose: To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (rs > 0) then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are greater than zero (sign bit is 0 but value not zero), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

None

Operations:

```
I:     tgt_offset ← sign_extend (offset || 0^2)
       condition ← GPR[rs] 63..0 > 0^GPRLEN
I+1:   if condition then
               PC ← PC + tgt_offset
       else
               NullifyCurrentInstruction()
       endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BLEZ — Branch on Less Than or Equal to Zero — BLEZ

31–26	25–21	20–16	15–0
BLEZ 000110	rs	0 00000	offset
6	5	5	16

MIPS I

Format: BLEZ rs, offset

Purpose: To test a GPR then do a PC-relative conditional branch.

Description: if (rs ≤ 0) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of the GPR *rs* are less than or equal to zero (sign bit is 1 or value is zero), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend (offset || 0^2)
      condition ← GPR[rs] 63..0 ≤ 0^GPRLEN
I+1:  if condition then
            PC ← PC + tgt_offset
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BLEZL Branch on Less Than or Equal to Zero Likely BLEZL

31 26	25 21	20 16	15 0
BLEZL 010110	rs	0 00000	offset
6	5	5	16

MIPS II

Format: BLEZL rs, offset

Purpose: To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (rs ≤ 0) then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than or equal to zero (sign bit is 1 or value is zero), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend (offset || 0^2)
             condition ← GPR[rs]63..0 ≤ 0^GPRLEN
I+1:  if condition then
             PC ← PC + tgt_offset
      else
             NullifyCurrentInstruction()
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BLTZ

Branch on Less Than Zero

31 – 26	25 – 21	20 – 16	15 – 0
REGIMM 000001	rs	BLTZ 00000	offset
6	5	5	16

MIPS I

Format: BLTZ rs, offset

Purpose: To test a GPR then do a PC-relative conditional branch.

Description: if (rs < 0) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

None

Operation:

```
I:     tgt_offset ← sign_extend (offset || 0^2)
              condition ← GPR[rs]_63..0 < 0^GPRLEN
I+1:   if condition then
              PC ← PC + tgt_offset
       endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BLTZAL Branch on Less Than Zero and Link BLTZAL

31–26	25–21	20–16	15–0
REGIMM 000001	rs	BLTZAL 10000	offset
6	5	5	16

MIPS I

Format: BLTZAL rs, offset

Purpose: To test a GPR then do a PC-relative conditional procedure call.

Description: if (rs < 0) then procedure_call

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch (**not** the branch itself), where execution would continue after a procedure call.

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch, in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

GPR 31 must not be used for the source register *rs*, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

Operation:

```
I:    tgt_offset ← sign_extend (offset || 0^2)
      condition ← GPR[rs]63..0 < 0^GPRLEN
      GPR[31]63..0 ← zero_extend (PC+8)
I+1:  if condition then
               PC ← PC + tgt_offset
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to more distant addresses.

BLTZALL Branch on Less Than Zero and Link Likely BLTZALL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZALL 10010	offset
6	5	5	16

MIPS II

Format: BLTZALL rs, offset

Purpose: To test a GPR then do a PC-relative conditional procedure call; execute the delay slot only if the branch is taken.

Description: if (rs < 0) then procedure_call_likely

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch (**not** the branch itself), where execution would continue after a procedure call.

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch, in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

GPR 31 must not be used for the source register *rs*, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

Operation:

```
I:    tgt_offset← sign_extend (offset || 0^2)
      condition ← GPR[rs] 63..0 < 0^GPRLEN
      GPR[31] 63..0 ← zero_extend (PC+8)
I+1:  if condition then
              PC ← PC + tgt_offset
      else
              NullifyCurrentInstruction()
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range ± 128 KB. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to more distant addresses.

BLTZL Branch on Less Than Zero Likely BLTZL

31–26	25–21	20–16	15–0
REGIMM 000001	rs	BLTZL 00010	offset
6	5	5	16

MIPS II

Format: BLTZL rs, offset

Purpose: To test a GPR then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (rs < 0) then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend (offset || 0^2)
      condition ← GPR[rs] 63..0 < 0^GPRLEN
I+1:  if condition then
              PC ← PC + tgt_offset
      else
              NullifyCurrentInstruction()
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BNE — Branch on Not Equal — BNE

31 … 26	25 … 21	20 … 16	15 … 0
BNE 000101	rs	rt	offset
6	5	5	16

MIPS I

Format: BNE rs, rt, offset

Purpose: To compare GPRs then do a PC-relative conditional branch.

Description: if (rs ≠ rt) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR rs and GPR rt are not equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend (offset || 0^2)
      condition ← (GPR[rs] 63..0 ≠ GPR[rt] 63..0)
I+1:  if condition then
              PC ← PC + tgt_offset
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BNEL — Branch on Not Equal Likely — BNEL

31–26	25–21	20–16	15–0
BNEL 010101	rs	rt	offset
6	5	5	16

MIPS II

Format: BNEL rs, rt, offset

Purpose: To compare GPRs then do a PC-relative conditional branch; execute the delay slot only if the branch is taken.

Description: if (rs ≠ rt) then branch_likely

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are not equal, branch to the effective target address after the instruction in the delay slot is executed. If the branch is not taken, the instruction in the delay slot is not executed.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend (offset || 0^2)
      condition ← (GPR[rs]63..0 ≠ GPR[rt]63..0)
I+1:  if condition then
              PC ← PC + tgt_offset
      else
              NullifyCurrentInstruction()
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BREAK Breakpoint BREAK

31 26	25 6	5 0
SPECIAL 000000	code	BREAK 001101
6	20	6

MIPS I

Format: BREAK

Purpose: To cause a Breakpoint exception.

Description:

A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler.

The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Restrictions:

None

Operation:

SignalException (Breakpoint)

Exceptions:

Breakpoint

Programming Notes:

None

DADD — Doubleword Add — DADD

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	DADD 101100
6	5	5	5	5	6

MIPS III

Format: DADD rd, rs, rt

Purpose: To add 64-bit integers. If overflow occurs, then trap.

Description: rd ← rs + rt

The 64-bit doubleword value in GPR *rt* is added to the 64-bit value in GPR *rs* to produce a 64-bit result. If the addition results in 64-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 64-bit result is placed into GPR *rd*.

Restrictions:

None

Operation:

```
temp ← GPR[rs] 63..0 + GPR[rt] 63..0
if (64_bit_arithmetic_overflow) then
    SignalException (IntegerOverflow)
else
    GPR[rd] 63..0 ← temp
endif
```

Exceptions:

Integer Overflow

Programming Notes:

DADDU performs the same arithmetic operation but, does not trap on overflow.

DADDI   Doubleword Add Immediate   DADDI

31 26	25 21	20 16	15 0
DADDI 011000	rs	rt	immediate
6	5	5	16

MIPS III

Format: DADDI rt, rs, immediate

Purpose: To add a constant to a 64-bit integer. If overflow occurs, then trap.

Description: rt ← rs + immediate

The 16-bit signed *immediate* is added to the 64-bit value in GPR *rs* to produce a 64-bit result. If the addition results in 64-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 64-bit result is placed into GPR *rt*.

Restrictions:

None

Operation:

```
temp ← GPR[rs] 63..0 + sign_extend (immediate)
if (64_bit_arithmetic_overflow) then
        SignalException (IntegerOverflow)
else
        GPR[rt] 63..0 ← temp
endif
```

Exceptions:

Integer Overflow

Programming Notes:

DADDIU performs the same arithmetic operation but, does not trap on overflow.

DADDIU

Doubleword Add Immediate Unsigned

DADDIU

31 ... 26	25 ... 21	20 ... 16	15 ... 0
DADDIU 011001	rs	rt	immediate
6	5	5	16

MIPS III

Format: DADDIU rt, rs, immediate

Purpose: To add a constant to a 64-bit integer.

Description: rt ← rs + immediate

The 16-bit signed *immediate* is added to the 64-bit value in GPR *rs* and the 64-bit arithmetic result is placed into GPR *rt.*

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

$GPR[rt]_{63..0} \leftarrow GPR[rs]_{63..0}$ + sign_extend (immediate)

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

DADDU

Doubleword Add Unsigned

DADDU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DADDU 101101
6	5	5	5	5	6

MIPS III

Format: DADDU rd, rs, rt

Purpose: To add 64-bit integers.

Description: rd ← rs + rt

The 64-bit doubleword value in GPR *rt* is added to the 64-bit value in GPR *rs* and the 64-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

$GPR[rd]_{63..0} \leftarrow GPR[rs]_{63..0} + GPR[rt]_{63..0}$

Exception:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

DIV Divide Word DIV

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DIV 011010
6	5	5	10	6

MIPS I

Format: DIV rs, rt

Purpose: To divide 32-bit signed integers.

Description: (LO, HI) ← rs / rt

The 32-bit word value in GPR *rs* is divided by the 32-bit value in GPR *rt*, treating both operands as signed values. The 32-bit quotient is placed into special register *LO* and the 32-bit remainder is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

If the divisor in GPR *rt* is zero, the arithmetic result value is undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
q ← $GPR[rs]_{31..0}$ div $GPR[rt]_{31..0}$
$LO_{63..0}$ ← sign_extend ($q_{31..0}$)
r ← $GPR[rs]_{31..0}$ mod $GPR[rt]_{31..0}$
$HI_{63..0}$ ← sign_extend ($r_{31..0}$)

Exceptions:

None

Supplementary Explanation:

Normally, when 0x80000000 (-2147483648) the signed minimum value is divided by 0xFFFFFFFF (-1), the operation will result in an overflow. However, in this instruction an overflow exception doesn’t occur and the result will be as follows:

Quotient is 0x80000000 (-2147483648), and remainder is 0x00000000 (0).

This sign of the quotient and the remainder is based on the signs of the dividend and the divisor as shown in the table below:

Dividend	Divisor	Quotient	Remainder
Positive	Positive	Positive	Positive
Positive	Negative	Negative	Positive
Negative	Positive	Negative	Negative
Negative	Negative	Positive	Negative

Programming Notes:

In the C790, the integer divide operation proceeds asynchronously and allows other CPU instructions to execute before it is retired. An attempt to read *LO* or *HI* before the results are written will wait (interlock) until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the divide so that other instructions can execute in parallel.

No arithmetic exception occurs under any circumstances. If divide-by-zero or overflow conditions should be detected and some action taken, then the divide instruction is typically followed by additional instructions to check for a zero divisor and / or for overflow. If the divide is asynchronous then the zero-divisor check can execute in parallel with the divide. The action taken on either divide-by-zero or overflow is either a convention within the program itself or more typically, the system software; one possibility is to take a BREAK exception with a code field value to signal the problem to the system software.

As an example, the C programming language in a UNIX environment expects division by zero to either terminate the program or execute a program-specified signal handler. C does not expect overflow to cause any exceptional condition. If the C compiler uses a divide instruction, it also emits code to test for a zero divisor and execute a BREAK instruction to inform the operating system if one is detected.

In the C790, sign-extended 32-bit values (bits 63..31) are ignored on divide operation.

DIVU

Divide Unsigned Word

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DIVU 011011
6	5	5	10	6

MIPS I

Format: DIVU rs, rt

Purpose: To divide 32-bit unsigned integers.

Description: (LO, HI) ← rs / rt

The 32-bit word value in GPR *rs* is divided by the 32-bit value in GPR *rt*, treating both operands as unsigned values. The 32-bit quotient is placed into special register *LO* and the 32-bit remainder is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

If the divisor in GPR *rt* is zero, the arithmetic result is undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
q ← (0 || $GPR[rs]_{31..0}$) div (0 || $GPR[rt]_{31..0}$)
$LO_{63..0}$ ← sign_extend ($q_{31..0}$)
r ← (0 || $GPR[rs]_{31..0}$) mod (0 || $GPR[rt]_{31..0}$)
$HI_{63..0}$ ← sign_extend ($r_{31..0}$)

Exceptions:

None

Programming Notes:

See the Programming Notes for the DIV instruction.

DSLL Doubleword Shift Left Logical DSLL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSLL 111000
6	5	5	5	5	6

MIPS III

Format: DSLL rd, rt, sa

Purpose: To left shift a doubleword by a fixed amount — 0 to 31 bits.

Description: rd ← rt << sa

The 64-bit doubleword contents of GPR *rt* are shifted left, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 0 to 31 is specified by *sa*.

Restrictions:

None

Operation:

s ← 0 || sa
$GPR[rd]_{63..0} \leftarrow GPR[rt]_{(63-s)..0} \| 0^s$

Exceptions:

None

Programming Notes:

None

DSLL32 Doubleword Shift Left Logical Plus 32 DSLL32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSLL32 111100
6	5	5	5	5	6

MIPS III

Format: DSLL32 rd, rt, sa

Purpose: To left shift a doubleword by a fixed amount — 32 to 63 bits.

Description: rd ← rt << (sa + 32)

The 64-bit doubleword contents of GPR *rt* are shifted left, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 32 to 63 is specified by *sa* + 32.

Restrictions:

None

Operation:

s ← 1 || sa /* 32 + sa */
$GPR[rd]_{63..0} \leftarrow GPR[rt]_{(63-s)..0} \parallel 0^s$

Exceptions:

None

Programming Notes:

None

DSLLV Doubleword Shift Left Logical Variable DSLLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSLLV 010100
6	5	5	5	5	6

MIPS III

Format: DSLLV rd, rt, rs

Purpose: To left shift a doubleword by a variable number of bits.

Description: rd ← rt << rs

The 64-bit doubleword contents of GPR *rt* are shifted left, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 0 to 63 is specified by the low-order six bits in GPR *rs*.

Restrictions:

None

Operation:

$s \leftarrow 0 \parallel GPR[rs]_{5..0}$
$GPR[rd]_{63..0} \leftarrow GPR[rt]_{(63-s)..0} \parallel 0^{s}$

Exceptions:

None

Programming Notes:

None

DSRA Doubleword Shift Right Arithmetic DSRA

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRA 111011
6	5	5	5	5	6

MIPS III

Format: DSRA rd, rt, sa

Purpose: To arithmetic right shift a doubleword by a fixed amount — 0 to 31 bits.

Description: rd ← rt >> sa (arithmetic)

The 64-bit doubleword contents of GPR *rt* are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 0 to 31 is specified by *sa*.

Restrictions:

None

Operation:

s ← 0 || sa
$GPR[rd]_{63..0} \leftarrow (GPR[rt]_{63})^{s} \,||\, GPR[rt]_{63..s}$

Exceptions:

None

Programming Notes:

None

DSRA32 Doubleword Shift Right Arithmetic Plus 32 DSRA32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRA32 111111
6	5	5	5	5	6

MIPS III

Format: DSRA32 rd, rt, sa

Purpose: To arithmetic right shift a doubleword by a fixed amount — 32-63 bits.

Description: rd ← rt >> (sa + 32) (arithmetic)

The doubleword contents of GPR *rt* are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 32 to 63 is specified by *sa* + 32.

Restrictions:

None

Operation:

s ←1 || sa /* 32 + sa */
$GPR[rd]_{63..0}$ ←$(GPR[rt]_{63})^{s}$ || $GPR[rt]_{63..s}$

Exceptions:

None

Programming Notes:

None

DSRAV Doubleword Shift Right Arithmetic Variable DSRAV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSRAV 010111
6	5	5	5	5	6

MIPS III

Format: DSRAV rd, rt, rs

Purpose: To arithmetic right shift a doubleword by a variable number of bits.

Description: rd ← rt >> rs (arithmetic)

The doubleword contents of GPR *rt* are shifted right, duplicating the sign bit (63) into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 0 to 63 is specified by the low-order six bits in GPR *rs*.

Restrictions:

None

Operation:

$s \leftarrow GPR[rs]_{5..0}$
$GPR[rd]_{63..0} \leftarrow (GPR[rt]_{63})^{s} \| GPR[rt]_{63..s}$

Exceptions:

None

Programming Notes:

None

DSRL

Doubleword Shift Right Logical

DSRL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRL 111010
6	5	5	5	5	6

MIPS III

Format: DSRL rd, rt, sa

Purpose: To logical right shift a doubleword by a fixed amount — 0 to 31 bits.

Description: rd ← rt >> sa (logical)

The doubleword contents of GPR *rt* are shifted right, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 0 to 31 is specified by *sa*.

Restrictions:

None

Operation:

$s \leftarrow 0 \parallel sa$
$GPR[rd]_{63..0} \leftarrow 0^{s} \parallel GPR[rt]_{63..s}$

Exceptions:

None

Programming Notes:

None

DSRL32 Doubleword Shift Right Logical Plus 32 DSRL32

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	0 00000	rt	rd	sa	DSRL32 111110
6	5	5	5	5	6

MIPS III

Format: DSRL32 rd, rt, sa

Purpose: To logical right shift a doubleword by a fixed amount — 32 to 63 bits.

Description: rd ← rt >> (sa + 32) (logical)

The 64-bit doubleword contents of GPR *rt* are shifted right, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 32 to 63 is specified by *sa* + 32.

Restrictions:

None

Operation:

s ← 1 || sa /* 32 + sa */
$GPR[rd]_{63..0}$ ← 0^s || $GPR[rt]_{63..s}$

Exceptions:

None

Programming Notes:

None

DSRLV Doubleword Shift Right Logical Variable DSRLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSRLV 010110
6	5	5	5	5	6

MIPS III

Format: DSRLV rd, rt, rs

Purpose: To logical right shift a doubleword by a variable number of bits.

Description: rd ← rt >> rs (logical)

The 64-bit doubleword contents of GPR *rt* are shifted right, inserting zeros into the emptied bits; the result is placed in GPR *rd*. The bit shift count in the range 0 to 63 is specified by the low-order six bits in GPR *rs*.

Restrictions:

None

Operation:

$s \leftarrow GPR[rs]_{5..0}$
$GPR[rd]_{63..0} \leftarrow 0^{s} \| GPR[rt]_{63..s}$

Exceptions:

None

Programming Notes:

None

DSUB

Doubleword Subtract

DSUB

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSUB 101110
6	5	5	5	5	6

MIPS III

Format: DSUB rd, rs, rt

Purpose: To subtract 64-bit integers; trap if overflow.

Description: rd ← rs - rt

The 64-bit doubleword value in GPR *rt* is subtracted from the 64-bit value in GPR *rs* to produce a 64-bit result. If the subtraction results in 64-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 64-bit result is placed into GPR *rd*.

Restrictions:

None

Operation:

```
temp ← GPR[rs] 63..0 - GPR[rt] 63..0
if (64_bit_arithmetic_overflow) then
        SignalException (IntegerOverflow)
else
        GPR[rd] 63..0 ← temp
endif
```

Exceptions:

Integer Overflow

Programming Notes:

DSUBU performs the same arithmetic operation but, does not trap on overflow.

DSUBU Doubleword Subtract Unsigned DSUBU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSUBU 101111
6	5	5	5	5	6

MIPS III

Format: DSUBU rd, rs, rt

Purpose: To subtract 64-bit integers.

Description: rd ← rs - rt

The 64-bit doubleword value in GPR *rt* is subtracted from the 64-bit value in GPR *rs* and the 64-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

$GPR[rd]_{63..0} \leftarrow GPR[rs]_{63..0} - GPR[rt]_{63..0}$

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

J Jump J

31 26	25 0
J 000010	instr_index
6	26

MIPS I

Format: J target

Purpose: To branch within the current 256 MB aligned region.

Description:

This is a PC-region branch (not PC-relative); the effective target address is in the "current" 256 MB aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (**not** the jump itself).

Jump to the effective target address. Execute the instruction following the jump, in the branch delay slot, before jumping.

Restrictions:

None

Operation:

I:
I+1: $PC \leftarrow PC_{31..28} \| instr_index \| 0^2$

Exceptions:

None

Programming Notes:

Forming the branch target address by concatenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch to anywhere in the region from anywhere in the region which a signed relative offset would not allow.

This definition creates the boundary case where the branch instruction is in the last word of a 256 MB region and can therefore only branch to the following 256 MB region containing the branch delay slot.

JAL Jump and Link JAL

31 26	25 0
JAL 000011	instr_index
6	26

MIPS I

Format: JAL target

Purpose: To procedure call within the current 256 MB aligned region.

Description:

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution would continue after a procedure call.

This is a PC-region branch (not PC-relative); the effective target address is in the "current" 256 MB aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (**not** the jump itself).

Jump to the effective target address. Execute the instruction following the jump, in the branch delay slot, before jumping.

Restrictions:

None

Operation:

I: $GPR[31]_{63..0} \leftarrow$ zero_extend (PC + 8)
I+1: $PC \leftarrow PC_{31..28}$ || instr_index || 0^2

Exceptions:

None

Programming Notes:

Forming the branch target address by concatenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch to anywhere in the region from anywhere in the region which a signed relative offset would not allow.

This definition creates the boundary case where the branch instruction is in the last word of a 256 MB region and can therefore only branch to the following 256 MB region containing the branch delay slot.

JALR

Jump and Link Register

31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0
SPECIAL 000000	rs	0 00000	rd	0 00000	JALR 001001
6	5	5	5	5	6

MIPS I

Format: JALR rs (rd = 31 implied)

JALR rd, rs

Purpose: To procedure call to an instruction address in a register.

Description: rd ← return_addr, PC ← rs

Place the return address link in GPR *rd*. The return link is the address of the second instruction following the branch, where execution would continue after a procedure call.

Jump to the effective target address in GPR *rs*. Execute the instruction following the jump, in the branch delay slot, before jumping.

Restrictions:

Register specifiers *rs* and *rd* must not be equal, because such an instruction does not have the same effect when re-executed. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by re-executing the branch when an exception occurs in the branch delay slot.

The effective target address in GPR *rs* must be naturally aligned. If either of the two least-significant bits are not -zero, then an Address Error exception occurs, not for the jump instruction, but when the branch target is subsequently fetched as an instruction.

Operation:

```
I:     temp ← GPR[rs] 31..0
       GPR[rd] 63..0 ← zero_extend (PC + 8)
I+1:   PC ← temp
```

Exceptions:

None

Programming Notes:

This is the only branch-and-link instruction that can select a register for the return link; all other link instructions use GPR 31 The default register for GPR *rd*, if omitted in the assembly language instruction, is GPR 31.

JR Jump Register JR

31 ... 26	25 ... 21	20 ... 6	5 ... 0
SPECIAL 000000	rs	0 000 0000 0000 0000	JR 001000
6	5	15	6

MIPS I

Format: JR rs

Purpose: To branch to an instruction address in a register.

Description: PC ← rs

Jump to the effective target address in GPR *rs*. Execute the instruction following the jump, in the branch delay slot, before jumping.

Restrictions:

The effective target address in GPR *rs* must be naturally aligned. If either of the two least-significant bits are not-zero, then an Address Error exception occurs, not for the jump instruction, but when the branch target is subsequently fetched as an instruction.

Operation:

I: temp ← $GPR[rs]_{31..0}$
I+1: PC ← temp

Exceptions:

None

Programming Notes:

None

LB Load Byte LB

31 26	25 21	20 16	15 0
LB 100000	base	rt	offset
6	5	5	16

MIPS I

Format: LB rt, offset (base)

Purpose: To load a byte from memory as a signed value.

Description: rt ← memory [base + offset]

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, sign-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + GPR[base] $_{31..0}$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr$_{(PSIZE-1)..4}$ || (pAddr$_{3..0}$ xor BigEndian^4)
memquad ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← vAddr$_{3..0}$ xor BigEndian^4
GPR[rt]$_{63..0}$ ← sign_extend (memquad$_{(7+8*byte)..8*byte}$)

Exceptions:

TLB Refill
TLB Invalid
Address Error

Programming Notes:

None

LBU

Load Byte Unsigned

31 26	25 21	20 16	15 0
LBU 100100	base	rt	offset
6	5	5	16

MIPS I

Format: LBU rt, offset (base)

Purpose: To load a byte from memory as an unsigned value.

Description: rt ← memory [base + offset]

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, zero-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + $GPR[base]_{31..0}$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{(PSIZE-1)..4}$ || ($pAddr_{3..0}$ xor $BigEndian^4$)
memquad ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← $vAddr_{3..0}$ xor $BigEndian^4$
$GPR[rt]_{63..0}$ ← zero_extend ($memquad_{(7+8*byte)..8*byte}$)

Exceptions:

TLB Refill
TLB Invalid
Address Error

Programming Notes:

None

LD — Load Doubleword — LD

31–26	25–21	20–16	15–0
LD 110111	base	rt	offset
6	5	5	16

MIPS III

Format: LD rt, offset (base)

Purpose: To load a doubleword from memory.

Description: rt ← memory [base + offset]

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If any of the three least-significant bits of the effective address are non-zero, an Address Error exception occurs.

Operation: (128-bit bus)

vAddr ←sign_extend (offset) + GPR [base]$_{31..0}$
if (vAddr$_{2..0}$) ≠ 0^3 then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr$_{(PSIZE-1)..4}$ || (pAddr$_{3..0}$ xor (BigEndian || 0^3))
byte ← vAddr$_{3..0}$ || (BigEndian || 0^3)
memquad ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
GPR[rt]$_{63..0}$ ← memquad$_{(63+8*byte)..8*byte}$

Exceptions:

TLB Refill
TLB Invalid
Address Error

Programming Notes:

None

LDL Load Doubleword Left LDL

31–26	25–21	20–16	15–0
LDL 011010	base	rt	offset
6	5	5	16

MIPS III

Format: LDL rt, offset (base)

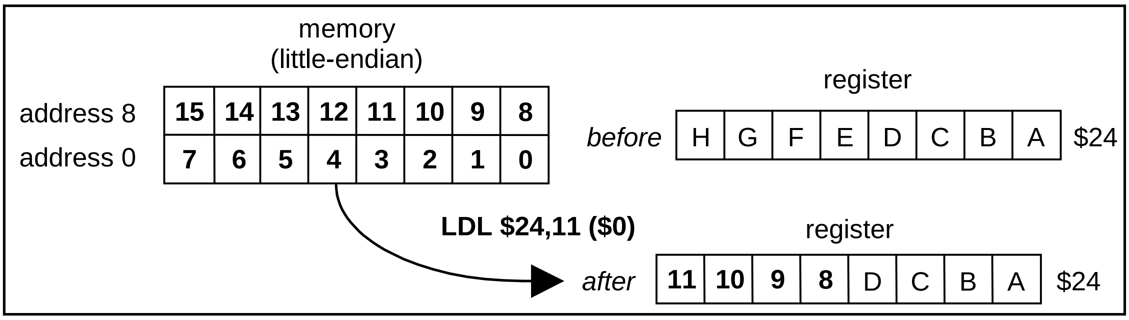
Purpose: To load the more-significant part of a doubleword from an unaligned memory address.

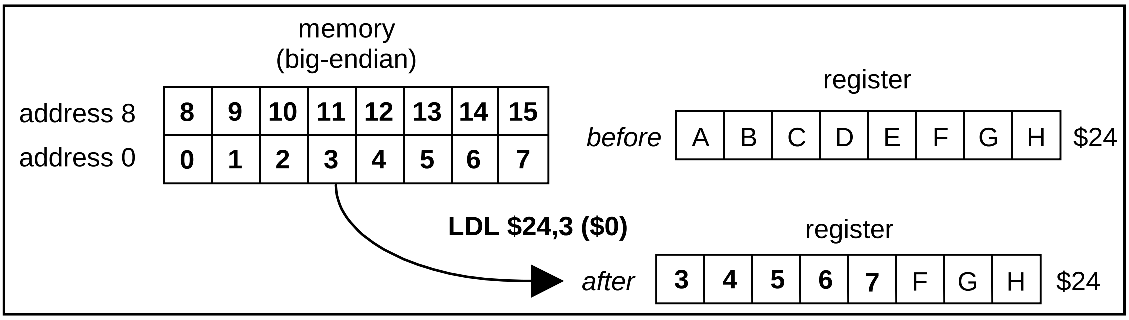
Description: rt ← rt MERGE memory [base + offset]

Paired LDL and LDR instructions are used to load a register with a doubleword from eight consecutive bytes in memory starting at an arbitrary byte address. LDL loads the left (most-significant) bytes and LDR loads the right (least-significant) bytes.

The instruction adds the 16-bit signed *offset* to the contents of GPR *base* to form the effective address. This is the address of the most-significant byte of a doubleword composed of eight consecutive bytes in memory. LDL loads from one to eight bytes, the most-significant bytes of the doubleword, into the corresponding bytes of GPR *rt*. It loads the bytes that are in the target doubleword that are also in the aligned doubleword which contains the byte specified by the effective address.

Conceptually, it starts at the specified byte in memory and loads that byte into the high-order (left-most) byte of the register; then it loads bytes from memory into the register until it reaches the low-order byte of the doubleword in memory. The least-significant (right-most) byte (s) of the register will not be changed.





The contents of GPR *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LDL (or LDR) instruction which also specifies register *rt*.

No address exceptions due to alignment are possible.

Restrictions:

None

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + $GPR[base]_{31..0}$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{(PSIZE-1)..4}$ || ($pAddr_{3..0}$ xor $BigEndian^4$)
if (BigEndian = 0) then
    pAddr ← $pAddr_{(PSIZE-1)..3}$ || 0^3
endif
byte ← 0 || ($vAddr_{2..0}$ xor $BigEndian^3$)
doubleword ← $vAddr_3$ xor BigEndian
memquad ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
$GPR[rt]_{63..0}$ ← $memquad_{(7+8*byte+64*doubleword)..(64*doubleword)}$ || $GPR[rt]_{(55-8*byte)..0}$

Given a doubleword in a register and a doubleword in memory, the operation of LDL is as follows:

LDL

	MSB 63							0 LSB
Register	a	b	c	d	e	f	g	h

Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Memory	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**

vAddr$_{3..0}$	Little-endian byte ordering (BigEndianCPU = 0)										
	Destination register contents after instruction(shaded is unchanged)								Type	offset	
	(63---------------------------------------32 31---0)									LEM	BEM
0	**X**	b	c	d	e	f	g	h	0	0	15
1	**W**	**X**	c	d	e	f	g	h	1	0	14
2	**V**	**W**	**X**	d	e	f	g	h	2	0	13
3	**U**	**V**	**W**	**X**	e	f	g	h	3	0	12
4	**T**	**U**	**V**	**W**	**X**	f	g	h	4	0	11
5	**S**	**T**	**U**	**V**	**W**	**X**	g	h	5	0	10
6	**R**	**S**	**T**	**U**	**V**	**W**	**X**	h	6	0	9
7	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	7	0	8
8	**P**	b	c	d	e	f	g	h	0	8	7
9	**O**	**P**	c	d	e	f	g	h	1	8	6
10	**N**	**O**	**P**	d	e	f	g	h	2	8	5
11	**M**	**N**	**O**	**P**	e	f	g	h	3	8	4
12	**L**	**M**	**N**	**O**	**P**	f	g	h	4	8	3
13	**K**	**L**	**M**	**N**	**O**	**P**	g	h	5	8	2
14	**J**	**K**	**L**	**M**	**N**	**O**	**P**	h	6	8	1
15	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	7	8	0

LDL

	MSB 63							0 LSB
Register	a	b	c	d	e	f	g	h

Big-endian	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Memory	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**
Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

vAddr3..0	Big-endian byte ordering (BigEndianCPU = 0)										
	Destination register contents after instruction(shaded is unchanged) (63------32 31------0)								Type	offset	
										LEM	BEM
0	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	7	0	0
1	**J**	**K**	**L**	**M**	**N**	**O**	**P**	h	6	0	1
2	**K**	**L**	**M**	**N**	**O**	**P**	g	h	5	0	2
3	**L**	**M**	**N**	**O**	**P**	f	g	h	4	0	3
4	**M**	**N**	**O**	**P**	e	f	g	h	3	0	4
5	**N**	**O**	**P**	d	e	f	g	h	2	0	5
6	**O**	**P**	c	d	e	f	g	h	1	0	6
7	**P**	b	c	d	e	f	g	h	0	0	7
8	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	7	8	8
9	**R**	**S**	**T**	**U**	**V**	**W**	**X**	h	6	8	9
10	**S**	**T**	**U**	**V**	**W**	**X**	g	h	5	8	10
11	**T**	**U**	**V**	**W**	**X**	f	g	h	4	8	11
12	**U**	**V**	**W**	**X**	e	f	g	h	3	8	12
13	**V**	**W**	**X**	d	e	f	g	h	2	8	13
14	**W**	**X**	c	d	e	f	g	h	1	8	14
15	**X**	b	c	d	e	f	g	h	0	8	15

LEM Little-endian memory (BigEndian = 0)
BEM BigEndian = 1
Type AccessLength sent to memory
Offset pAddr3..0 sent to memory

Exceptions:

TLB Refill
TLB Invalid
Address Error

Programming Notes:

None

LDR

Load Doubleword Right

31 26	25 21	20 16	15 0
LDR 011011	base	rt	offset
6	5	5	16

MIPS III

Format: LDR rt, offset (base)

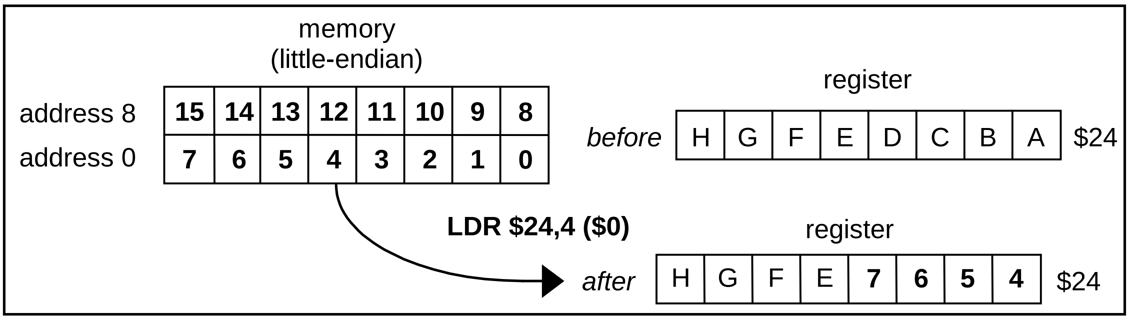
Purpose: To load the less-significant part of a doubleword from an unaligned memory address.

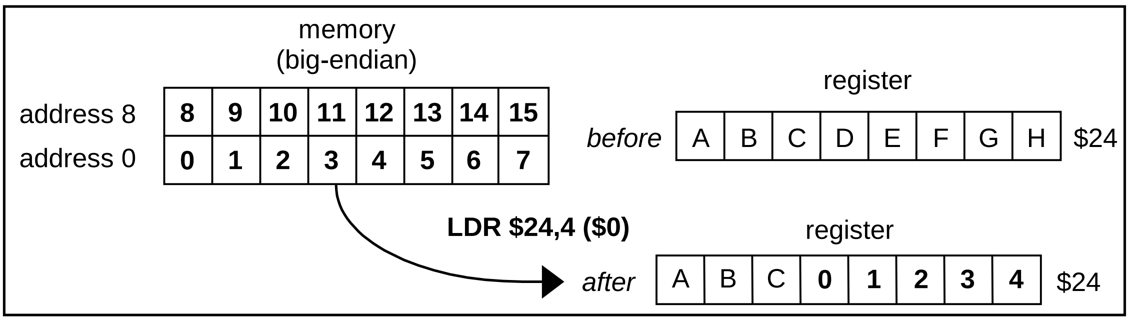
Description: rt ← rt MERGE memory [base + offset]

Paired LDL and LDR instructions are used to load a register with a doubleword from eight consecutive bytes in memory starting at an arbitrary byte address. LDL loads the left (most-significant) bytes and LDR loads the right (least-significant) bytes.

The instruction adds the 16-bit signed *offset* to the contents of GPR *base* to form the effective address. This is the address of the least-significant bytes of a doubleword composed of eight consecutive bytes in memory. LDR loads from one to eight bytes, the least-significant bytes of the doubleword, into the corresponding bytes of GPR *rt*. It loads the bytes that are in the target doubleword that are also in the aligned doubleword which contains the byte specified by the effective address.

Conceptually, it starts at the specified byte in memory and loads that byte into the low-order (right-most) byte of the register; then it loads bytes from memory into the register until it reaches the high-order byte of the doubleword in memory. The most significant (left-most) byte (s) of the register will not be changed.





The contents of GPR *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LDR (or LDL) instruction which also specifies register *rt*.

No address exceptions due to alignment are possible.

Restrictions:

None

Operation: (128-bit bus)

$vAddr \leftarrow sign_extend(offset) + GPR[base]_{31..0}$
$(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA, LOAD)$
$pAddr \leftarrow pAddr_{(PSIZE-1)..0} \parallel (pAddr_{3..0}\ xor\ BigEndian^4)$
if (BigEndian = 1) then
  $pAddr \leftarrow pAddr_{(PSIZE-1)..3} \parallel 0^3$
endif
$byte \leftarrow 0 \parallel (vAddr_{2..0}\ xor\ BigEndian^3)$
$doubleword \leftarrow vAddr_3\ xor\ BigEndian$
$memquad \leftarrow LoadMemory\ (uncached, byte, pAddr, vAddr, DATA)$
$GPR[rt]_{63..0} \leftarrow GPR[rt]_{63..(64-8*byte)} \parallel memquad_{(63+64*doubleword)..(64*doubleword+8*byte)}$

Given a doubleword in a register and a doubleword in memory, the operation of LDR is as follows:

LDR

	MSB 63							0 LSB
Register	a	b	c	d	e	f	g	h

Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Memory	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**

$vAddr_{3..0}$	Little-endian byte ordering (BigEndianCPU = 0)										
	Destination register contents after instruction(shaded is unchanged)								Type	offset	
	(63---------------------------------------32 31--0)									LEM	BEM
0	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	7	0	0
1	a	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	6	1	0
2	a	b	**Q**	**R**	**S**	**T**	**U**	**V**	5	2	0
3	a	b	c	**Q**	**R**	**S**	**T**	**U**	4	3	0
4	a	b	c	d	**Q**	**R**	**S**	**T**	3	4	0
5	a	b	c	d	e	**Q**	**R**	**S**	2	5	0
6	a	b	c	d	e	f	**Q**	**R**	1	6	0
7	a	b	c	d	e	f	g	**Q**	0	7	0
8	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	7	8	0
9	a	**I**	**J**	**K**	**L**	**M**	**N**	**O**	6	9	0
10	a	b	**I**	**J**	**K**	**L**	**M**	**N**	5	10	0
11	a	b	c	**I**	**J**	**K**	**L**	**M**	4	11	0
12	a	b	c	d	**I**	**J**	**K**	**L**	3	12	0
13	a	b	c	d	e	**I**	**J**	**K**	2	13	0
14	a	b	c	d	e	f	**I**	**J**	1	14	0
15	a	b	c	d	e	f	g	**I**	0	15	0

LDR

	MSB 63							0 LSB
Register	a	b	c	d	e	f	g	h

Big-endian	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Memory	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**
Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

vAddr$_{3..0}$	Big-endian byte ordering (BigEndianCPU = 1)										
	Destination register contents after instruction(shaded is unchanged) (63--32 31--0)								Type	offset	
										LEM	BEM
0	**a**	**b**	**c**	**d**	**e**	**f**	**g**	I	0	15	0
1	**a**	**b**	**c**	**d**	**e**	**f**	I	J	1	14	0
2	**a**	**b**	**c**	**d**	**e**	I	J	K	2	13	0
3	**a**	**b**	**c**	**d**	I	J	K	L	3	12	0
4	**a**	**b**	**c**	I	J	K	L	M	4	11	0
5	**a**	**b**	**I**	J	K	L	M	N	5	10	0
6	**a**	I	J	K	L	M	N	O	6	9	0
7	I	J	K	L	M	N	O	P	7	8	0
8	**a**	**b**	**c**	**d**	**e**	f	g	Q	0	7	0
9	**a**	**b**	**c**	**d**	**e**	f	Q	R	1	6	0
10	**a**	**b**	**c**	**d**	**e**	Q	R	S	2	5	0
11	**a**	**b**	**c**	**d**	Q	R	S	T	3	4	0
12	**a**	**b**	**c**	Q	R	S	T	U	4	3	0
13	**a**	**b**	Q	R	S	T	U	V	5	2	0
14	**a**	Q	R	S	T	U	V	W	6	1	0
15	Q	R	S	T	U	V	W	**X**	7	0	0

LEM Little-endian memory (BigEndianMem = 0)
BEM BigEndianMem = 1
Type AccessLength sent to memory
Offset pAddr$_{2..0}$ sent to memory

Exceptions:

TLB Refill
TLB Invalid
Address Error

Programming Notes:

None

LH   Load Halfword   LH

31 — 26	25 — 21	20 — 16	15 — 0
LH 100001	base	rt	offset
6	5	5	16

MIPS I

Format: LH rt, offset (base)

Purpose: To load a halfword from memory as a signed value.

Description: rt ← memory [base + offset]

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, sign-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + GPR[base] $_{31..0}$
if ($vAddr_0$) ≠ 0 then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{(PSIZE-1)..4}$ || ($pAddr_{3..0}$ xor ($BigEndian^3$ || 0))
memquad ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
byte ← $vAddr_{3..0}$ xor ($BigEndian^3$ || 0)
$GPR[rt]_{63..0}$ ← sign_extend ($memquad_{(15+8*byte)..8*byte}$)

Exceptions:

TLB Refill
TLB Invalid
Address Error

Programming Notes:

None

LHU Load Halfword Unsigned LHU

31 26	25 21	20 16	15 0
LHU 100101	base	rt	offset
6	5	5	16

MIPS I

Format: LHU rt, offset (base)

Purpose: To load a halfword from memory as an unsigned value.

Description: rt ← memory [base + offset]

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, zero-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + GPR [base] $_{31..0}$
if (vAddr$_0$) ≠ 0 then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr$_{(PSIZE-1)..4}$ || (pAddr$_{3..0}$ xor (BigEndian^3 || 0))
memquad ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
byte ← vAddr$_{3..0}$ xor (BigEndian^3 || 0)
GPR [rt]$_{63..0}$ ← zero_extend (memquad$_{(15+8*byte)..8*byte}$)

Exceptions:

TLB Refill
TLB Invalid
Address Error

Programming Notes:

None

LUI

Load Upper Immediate

31–26	25–21	20–16	15–0
LUI 001111	0 00000	rt	immediate
6	5	5	16

MIPS I

Format: LUI rt, immediate

Purpose: To load a constant into the upper half of a word.

Description: rt ← immediate || 0^{16}

The 16-bit *immediate* is shifted left 16 bits and concatenated with 16 bits of low-order zeros. The 32-bit result is sign-extended and placed into GPR *rt*.

Restrictions:

None

Operation:

GPR [rt]$_{63..0}$ ← sign_extend (immediate || 0^{16})

Exceptions:

None

Programming Notes:

None

LW Load Word LW

31 26	25 21	20 16	15 0
LW 100011	base	rt	offset
6	5	5	16

MIPS I

Format: LW rt, offset (base)

Purpose: To load a word from memory as a signed value.

Description: rt ← memory [base + offset]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If either of the two least-significant bits of the address are non-zero, an Address Error exception occurs.

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + GPR [base] $_{31..0}$
if (vAddr$_{1..0}$) ≠ 0^2 then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr$_{(PSIZE-1)..4}$ || (pAddr$_{3..0}$ xor (BigEndian^2 || 0^2))
memquad ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
byte ← vAddr$_{3..0}$ xor (BigEndian^2 || 0^2)
GPR [rt] $_{63..0}$ ← sign_extend (memquad$_{(31+8*byte)..8*byte}$)

Exceptions:

TLB Refill
TLB Invalid
Address Error

Programming Notes:

None

LWL Load Word Left LWL

31 26	25 21	20 16	15 0
LWL 100010	base	rt	offset
6	5	5	16

MIPS I

Format: LWL rt, offset (base)

Purpose: To load the more-significant part of a word from an unaligned memory address as a signed value.

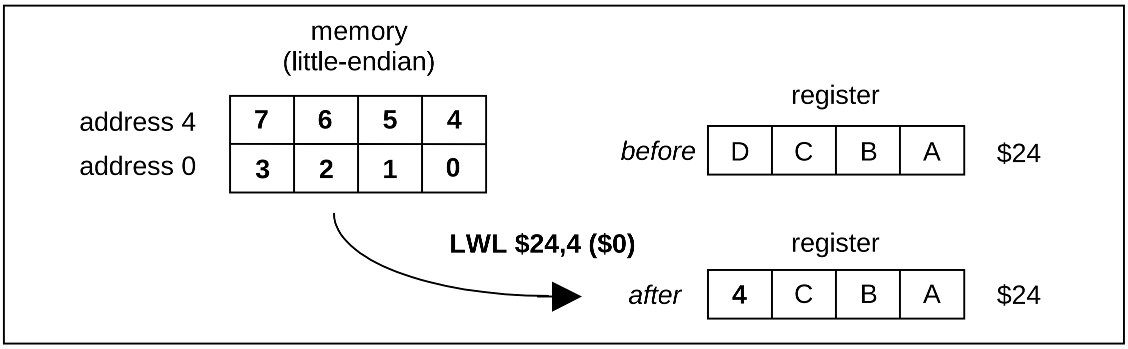
Description: rt ← rt MERGE memory [base + offset]

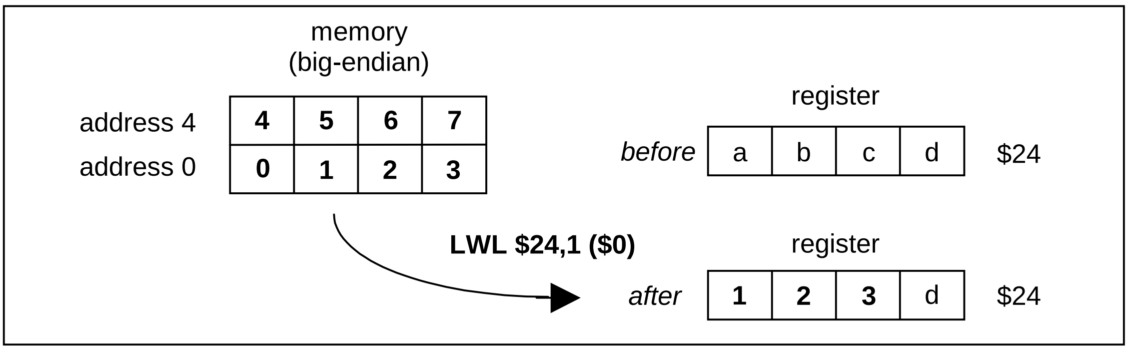
Paired LWL and LWR instructions are used to load a register with a word from four consecutive bytes in memory starting at an arbitrary byte address. LWL loads the left (most-significant) bytes and LWR loads the right (least-significant) bytes.

The instruction adds the 16-bit signed *offset* to the contents of GPR *base* to form the effective address. This is the address of the most-significant byte of a word composed of four consecutive bytes in memory. LWL loads from one to four bytes, the most-significant bytes of the word, into the corresponding bytes of GPR *rt*. It loads the bytes that are in the target word that are also in the aligned word which contains the byte specified by the effective address.

Bit 31 of the register is loaded so the loaded word is sign-extended.

Conceptually, it starts at the specified byte in memory and loads that byte into the high-order (left-most) byte of the register; then it loads bytes from memory into the register until it reaches the low-order byte of the word in memory. The least-significant (right-most) byte(s) of the register will not be changed.





The contents of GPR *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LWL (or LWR) instruction which also specifies register *rt*.

No address exceptions due to alignment are possible.

Restrictions:

None

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + GPR [base]$_{31..0}$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr$_{(PSIZE-1)..4}$ || (pAddr$_{3..0}$ xor BigEndian^4)
if (BigEndian = 0) then
  pAddr$_{(PSIZE-1)..3}$ || 0^3
endif
byte ← 0^2 || (vAddr$_{1..0}$ xor BigEndian^2)
word ← vAddr$_{3..2}$ xor BigEndian^2
memquad ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
temp ← memquad$_{(32*word+8*byte+7)..32*word}$ || GPR [rt]$_{(23-8*byte)..0}$
GPR [rt]$_{63..0}$ ← (temp$_{31}$)^{32} || temp

Given a doubleword in a register and a doubleword in memory, the operation of LWL is as follows:

LWL

	MSB 63							0 LSB
Register	a	b	c	d	e	f	g	h

Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Memory	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**

vAddr$_{3..0}$	Little-endian byte ordering (BigEndianCPU = 0)							
	Destination register contents after instruction(shaded is unchanged)					Type	offset	
	(63--32 31--0)						LEM	BEM
0	Sign bit(31) extended	**X**	f	g	h	0	0	15
1	Sign bit(31) extended	**W**	**X**	g	h	1	0	14
2	Sign bit(31) extended	**V**	**W**	**X**	h	2	0	13
3	Sign bit(31) extended	**U**	**V**	**W**	**X**	3	0	12
4	Sign bit(31) extended	**T**	f	g	h	0	4	11
5	Sign bit(31) extended	**S**	**T**	g	h	1	4	10
6	Sign bit(31) extended	**R**	**S**	**T**	h	2	4	9
7	Sign bit(31) extended	**Q**	**R**	**S**	**T**	3	4	8
8	Sign bit(31) extended	**P**	f	g	h	0	8	7
9	Sign bit(31) extended	**O**	**P**	g	h	1	8	6
10	Sign bit(31) extended	**N**	**O**	**P**	h	2	8	5
11	Sign bit(31) extended	**M**	**N**	**O**	**P**	3	8	4
12	Sign bit(31) extended	**L**	f	g	h	0	12	3
13	Sign bit(31) extended	**K**	**L**	g	h	1	12	2
14	Sign bit(31) extended	**J**	**K**	**L**	h	2	12	1
15	Sign bit(31) extended	**I**	**J**	**K**	**L**	3	12	0

LWL

	MSB 63							0 LSB
Register	a	b	c	d	e	f	g	h

Big-endian	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Memory	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**
Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

vAddr$_{3..0}$	Big-endian byte ordering (BigEndianCPU = 1)							
	Destination register contents after instruction(shaded is unchanged) (63---------------------------------------32 31---0)					Type	offset	
							LEM	BEM
0	Sign bit(31) extended	**I**	**J**	**K**	**L**	3	12	0
1	Sign bit(31) extended	**J**	**K**	**L**	h	2	12	1
2	Sign bit(31) extended	**K**	**L**	g	h	1	12	2
3	Sign bit(31) extended	**L**	f	g	h	0	12	3
4	Sign bit(31) extended	**M**	**N**	**O**	**P**	3	8	4
5	Sign bit(31) extended	**N**	**O**	**P**	h	2	8	5
6	Sign bit(31) extended	**O**	**P**	g	h	1	8	6
7	Sign bit(31) extended	**P**	f	g	h	0	8	7
8	Sign bit(31) extended	**Q**	**R**	**S**	**T**	3	4	8
9	Sign bit(31) extended	**R**	**S**	**T**	h	2	4	9
10	Sign bit(31) extended	**S**	**T**	g	h	1	4	10
11	Sign bit(31) extended	**T**	f	g	h	0	4	11
12	Sign bit(31) extended	**U**	**V**	**W**	**X**	3	0	12
13	Sign bit(31) extended	**V**	**W**	**X**	h	2	0	13
14	Sign bit(31) extended	**W**	**X**	g	h	1	0	14
15	Sign bit(31) extended	**X**	f	g	h	0	0	15

LEM Little-endian memory (BigEndianMem = 0)
BEM BigEndianMem = 1
Type AccessLength sent to memory
Offset pAddr$_{2..0}$ sent to memory

Exceptions:

TLB Refill
TLB Invalid
Address Error

Programming Notes:

The architecture provides no direct support for treating unaligned words as unsigned values, i.e. zeroing bits 63..32 of the destination register when bit 31 is loaded. See SLL or SLLV for a single-instruction method of propagating the word sign bit in a register into the upper half of a 64-bit register.

LWR Load Word Right LWR

31 ... 26	25 ... 21	20 ... 16	15 ... 0
LWR 100110	base	rt	offset
6	5	5	16

MIPS I

Format: LWR rt, offset (base)

Purpose: To load the less-significant part of a word from an unaligned memory address as a signed value.

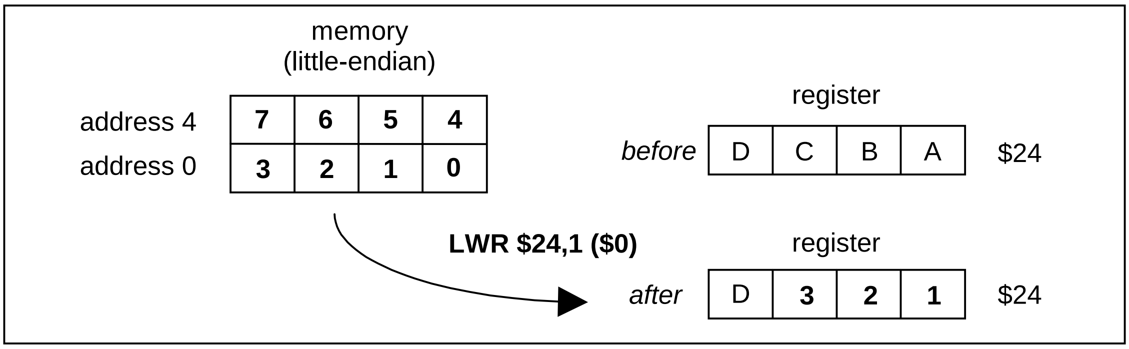
Description: rt ← rt MERGE memory [base + offset]

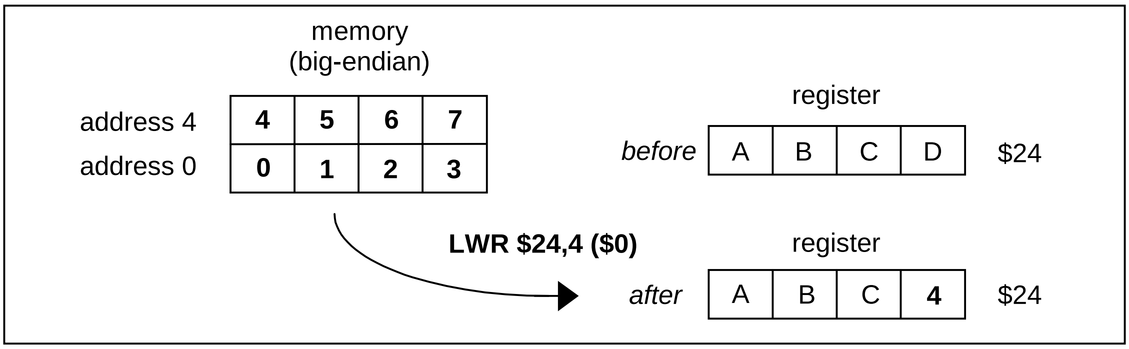
Paired LWL and LWR instructions are used to load a register with a word from four consecutive bytes in memory starting at an arbitrary byte address. LWL loads the left (most-significant) bytes and LWR loads the right (least-significant) bytes.

The instruction adds the 16-bit signed *offset* to the contents of GPR *base* to form the effective address. This is the address of the least-significant byte of a word composed of four consecutive bytes in memory. LWR loads from one to four bytes, the least-significant bytes of the word, into the corresponding bytes of GPR *rt*. It loads the bytes that are in the target word that are also in the aligned word which contains the byte specified by the effective address.

If the word sign bit (bit 31) is loaded from memory into the register by the instruction, then the loaded word is sign-extended. If the sign bit is not loaded from memory by the LWR, then bits 63..32 of the destination are unchanged.

Conceptually, it starts at the specified byte in memory and loads that byte into the low-order (right-most) byte of the register; then it loads bytes from memory into the register until it reaches the high-order byte of the word in memory. The most significant (left-most) byte(s) of the register will not be changed.





The contents of GPR *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LWR (or LWL) instruction which also specifies register *rt*.

No address exceptions due to alignment are possible.

Restrictions:

None

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + GPR [base]$_{31..0}$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr$_{(PSIZE-1)..4}$ || (pAddr$_{3..0}$ xor BigEndian^{4})
if (BigEndian = 1) then
  pAddr$_{(PSIZE-31)..3}$ || 0^{3}
endif
byte ← 0 || (vAddr$_{1..0}$ xor BigEndian^{2})
word ← vAddr$_{3..2}$ xor BigEndian^{2}
memquad ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
temp ← GPR [rt]$_{31..(32-8*byte)}$ || memquad$_{(31+32*word)..(32*word+8*byte)}$
if (byte = 4) then
    utemp ← (temp$_{31}$)^{32} /* loaded bit 31, must sign extend */
else
  one of the following two behaviors:
    utemp ← GPR [rt]$_{63..32}$ /* leave what was there alone */
    utemp ← (GPR [rt]$_{31}$)^{32} /* sign-extend bit 31 */
endif
GPR [rt]$_{63..0}$ ← utemp || temp

Given a word in a register and a word in memory, the operation of LWR is as follows:

LWR

	MSB 63							0 LSB
Register	a	b	c	d	e	f	g	h

Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Memory	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**

$vAddr_{3..0}$	Little-endian byte ordering (BigEndianCPU = 0)							
	Destination register contents after instruction(shaded is unchanged) (63--32 31--0)					Type	offset	
							LEM	BEM
0	Sign bit (31) extended	e	f	g	**I**	0	15	0
1	Sign bit (31) extended or unchanged	e	f	**I**	**J**	1	14	0
2	Sign bit (31) extended or unchanged	e	**I**	**J**	**K**	2	13	0
3	Sign bit (31) extended or unchanged	**I**	**J**	**K**	**L**	3	12	0
4	Sign bit (31) extended	e	f	g	**M**	0	11	4
5	Sign bit (31) extended or unchanged	e	f	**M**	**N**	1	10	4
6	Sign bit (31) extended or unchanged	e	**M**	**N**	**O**	2	9	4
7	Sign bit (31) extended or unchanged	**M**	**N**	**O**	**P**	3	8	4
8	Sign bit (31) extended	e	f	g	**Q**	0	7	8
9	Sign bit (31) extended or unchanged	e	f	**Q**	**R**	1	6	8
10	Sign bit (31) extended or unchanged	e	**Q**	**R**	**S**	2	5	8
11	Sign bit (31) extended or unchanged	**Q**	**R**	**S**	**T**	3	4	8
12	Sign bit (31) extended	e	f	g	**U**	0	3	12
13	Sign bit (31) extended or unchanged	e	f	**U**	**V**	1	2	12
14	Sign bit (31) extended or unchanged	e	**U**	**V**	**W**	2	1	12
15	Sign bit (31) extended or unchanged	**U**	**V**	**W**	**X**	3	0	12

LWR

	MSB 63							0 LSB
Register	a	b	c	d	e	f	g	h

Big-endian	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Memory	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**
Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

$vAddr_{3..0}$	Big-endian byte ordering (BigEndianCPU = 1)							
	Destination register contents after instruction(shaded is unchanged) (63--32 31--0)					Type	offset	
							LEM	BEM
0	Sign bit (31) extended or unchanged	e	f	g	**I**	0	15	0
1	Sign bit (31) extended or unchanged	e	f	**I**	**J**	1	14	0
2	Sign bit (31) extended or unchanged	e	**I**	**J**	**K**	2	13	0
3	Sign bit (31) extended	**I**	**J**	**K**	**L**	3	12	0
4	Sign bit (31) extended or unchanged	e	f	g	**M**	0	11	4
5	Sign bit (31) extended or unchanged	e	f	**M**	**N**	1	10	4
6	Sign bit (31) extended or unchanged	e	**M**	**N**	**O**	2	9	4
7	Sign bit (31) extended	**M**	**N**	**O**	**P**	3	8	4
8	Sign bit (31) extended or unchanged	e	f	g	**Q**	0	7	8
9	Sign bit (31) extended or unchanged	e	f	**Q**	**R**	1	6	8
10	Sign bit (31) extended or unchanged	e	**Q**	**R**	**S**	2	5	8
11	Sign bit (31) extended	**Q**	**R**	**S**	**T**	3	4	8
12	Sign bit (31) extended or unchanged	e	f	g	**U**	0	3	12
13	Sign bit (31) extended or unchanged	e	f	**U**	**V**	1	2	12
14	Sign bit (31) extended or unchanged	e	**U**	**V**	**W**	2	1	12
15	Sign bit (31) extended	**U**	**V**	**W**	**X**	3	0	12

LEM Little-endian memory (BigEndian = 0)
BEM BigEndianMem = 1
Type AccessLength sent to memory
Offset $pAddr_{2..0}$ sent to memory

Exceptions:

TLB Refill
TLB Invalid
Address Error

Programming Notes:

The architecture provides no direct support for treating unaligned words as unsigned values, i.e. zeroing bits 63..32 of the destination register when bit 31 is loaded. See SLL or SLLV for a single-instruction method of propagating the word sign bit in a register into the upper half of a 64-bit register.

LWU

Load Word Unsigned

31 26	25 21	20 16	15 0
LWU 100111	base	rt	offset
6	5	5	16

MIPS III

Format: LWU rt, offset (base)

Purpose: To load a word from memory as an unsigned value.

Description: rt ← memory [base + offset]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, zero-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If either of the two least-significant bits of the address are non-zero, an Address Error Exception occurs.

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + GPR [base] $_{31..0}$
if (vAddr$_{1..0}$) ≠ 0^2 then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr$_{(PSIZE-1)..4}$ || (pAddr$_{3..0}$ xor (BigEndian^2 || 0^2))
memquad ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
byte ← vAddr$_{3..0}$ xor (BigEndian^2 || 0^2)
GPR [rt] $_{63..0}$ ← 0^{32} || memquad$_{(31+8*byte)..8*byte}$

Exceptions:

TLB Refill
TLB Invalid
Address Error

Programming Notes:

None

MFHI

Move from HI Register

31 — 26	25 — 16	15 — 11	10 — 6	5 — 0
SPECIAL 000000	0 00 0000 0000	rd	0 00000	MFHI 010000
6	10	5	5	6

MIPS I

Format: MFHI rd

Purpose: To copy the special purpose HI register to a GPR.

Description: rd ← HI

The contents of special register *HI* are loaded into GPR *rd*.

Restrictions:

None

Operation:

GPR $[rd]_{63..0} \leftarrow HI_{63..0}$

Exceptions:

None

Programming Notes:

No restriction is needed because C790 has an interlock mechanism for MULT or DIV instructions.

MFLO Move from LO Register MFLO

31 26	25 16	15 11	10 6	5 0
SPECIAL 000000	0 00 0000 0000	rd	0 00000	MFLO 010010
6	10	5	5	6

MIPS I

Format: MFLO rd

Purpose: To copy the special purpose LO register to a GPR.

Description: rd ← LO

The contents of special register *LO* are loaded into GPR *rd*.

Restrictions:

None

Operation:

GPR [rd] $_{63..0}$ ← LO$_{63..0}$

Exceptions:

None

Programming Notes:

(Same as MFHI)

MOVN — Move Conditional on Not Zero — MOVN

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	MOVN 001011
6	5	5	5	5	6

MIPS IV

Format: MOVN rd, rs, rt

Purpose: To conditionally move a GPR after testing a GPR value.

Description: if (rt ≠ 0) then rd ← rs

If the value in GPR *rt* is not equal to zero, then the contents of GPR *rs* are placed into GPR *rd*.

Restrictions:

None

Operation:

if GPR [rt] $_{63..0}$ ≠ 0 then
    GPR [rd] $_{63..0}$ ← GPR [rs] $_{63..0}$
endif

Exceptions:

None

Programming Notes:

The nonzero value tested here is the “condition true” result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

MOVZ

Move Conditional on Zero

MOVZ

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	MOVZ 001010
6	5	5	5	5	6

MIPS IV

Format: MOVZ rd, rs, rt

Purpose: To conditionally move a GPR after testing a GPR value.

Description: if (rt = 0) then rd ← rs

If the value in GPR *rt* is equal to zero, then the contents of GPR *rs* are placed into GPR *rd*.

Restrictions:

None

Operation:

```
if GPR [rt] 63..0 = 0 then
        GPR [rd] 63..0 ← GPR [rs] 63..0
endif
```

Exceptions:

None

Programming Notes:

The zero value tested here is the “condition false” result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

MTHI Move to HI Register MTHI

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 000 0000 0000 0000	MTHI 010001
6	5	15	6

MIPS I

Format: MTHI rs

Purpose: To copy a GPR to the special purpose HI register.

Description: HI ← rs

The contents of GPR *rs* are loaded into special register *HI*.

Restrictions:

None

Operation:

$HI_{63..0} \leftarrow GPR[rs]_{63..0}$

Exceptions:

None

Programming Notes:

None

MTLO Move to LO Register MTLO

31–26	25–21	20–6	5–0
SPECIAL 000000	rs	0 000 0000 0000 0000	MTLO 010011
6	5	15	6

MIPS I

Format: MTLO rs

Purpose: To copy a GPR to the special purpose LO register.

Description: LO ← rs

The contents of GPR *rs* are loaded into special register *LO*.

Restrictions:

None

Operation:

$LO_{63..0} \leftarrow GPR[rs]_{63..0}$

Exceptions:

None

Programming Notes:

None

MULT

Multiply Word

MULT

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	MULT 011000
6	5	5	10	6

MIPS I

Format: MULT rs, rt

Purpose: To multiply 32-bit signed integers.

Description: (LO, HI) ← rs × rt

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as signed values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register *LO*, and the high-order 32-bit word is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

if (NotWordValue (GPR [rs]) or NotWordValue (GPR [rt])) then UndefinedResult() endif
prod ← GPR $[rs]_{31..0}$ * GPR $[rt]_{31..0}$
$LO_{63..0}$ ← $(prod_{31})^{32}$ || $prod_{31..0}$
$HI_{63..0}$ ← $(prod_{63})^{32}$ || $prod_{63..32}$

Exceptions:

None

Programming Notes:

In the C790, the integer multiply operation proceeds asynchronously and allows other CPU instructions to execute before it is retired. An attempt to read *LO* or *HI* before the results are written will wait (interlock) until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

MULTU   Multiply Unsigned Word   MULTU

31 … 26	25 … 21	20 … 16	15 … 6	5 … 0
SPECIAL 000000	rs	rt	0 00 0000 0000	MULTU 011001
6	5	5	10	6

MIPS I

Format: MULTU rs, rt

Purpose: To multiply 32-bit unsigned integers.

Description: (LO, HI) ← rs × rt

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as unsigned values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register *LO*, and the high-order 32-bit word is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

if (NotWordValue (GPR [rs]) or NotWordValue (GPR [rt])) then UndefinedResult() endif
prod ← (0 || GPR $[rs]_{31..0}$) * (0 || GPR $[rt]_{31..0}$)
$LO_{63..0}$ ← $(prod_{31})^{32}$ || $prod_{31..0}$
$HI_{63..0}$ ← $(prod_{63})^{32}$ || $prod_{63..32}$

Exceptions:

None

Programming Notes:

See the Programming Notes for the MULT instruction.

NOR Not Or NOR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	NOR 100111
6	5	5	5	5	6

MIPS I

Format: NOR rd, rs, rt

Purpose: To do a bitwise logical NOT OR.

Description: rd ← rs NOR rt

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical NOR operation. The result is placed into GPR *rd*.

Restrictions:

None

Operation:

GPR [rd] $_{63..0}$ ← GPR [rs] $_{63..0}$ nor GPR [rt] $_{63..0}$

Exceptions:

None

Programming Notes:

None

OR

Or

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	OR 100101
6	5	5	5	5	6

MIPS I

Format: OR rd, rs, rt

Purpose: To do a bitwise logical OR.

Description: rd ← rs OR rt

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical OR operation. The result is placed into GPR *rd*.

Restrictions:

None

Operation:

$GPR [rd]_{63..0} \leftarrow GPR [rs]_{63..0}$ or $GPR [rt]_{63..0}$

Exceptions:

None

Programming Notes:

None

ORI Or Immediate ORI

31 — 26	25 — 21	20 — 16	15 — 0
ORI 001101	rs	rt	immediate
6	5	5	16

MIPS I

Format: ORI rt, rs, immediate

Purpose: To do a bitwise logical OR with a constant.

Description: rt ← rs OR immediate

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR *rs* in a bitwise logical OR operation. The result is placed into GPR *rt*.

Restrictions:

None

Operation:

$GPR[rt]_{63..0} \leftarrow$ zero_extend (immediate) or $GPR[rs]_{63..0}$

Exceptions:

None

Programming Notes:

None

PREF Prefetch PREF

31 26	25 21	20 16	15 0
PREF 110011	base	hint	offset
6	5	5	16

MIPS IV

Format: PREF hint, offset (base)

Purpose: To prefetch data from memory.

Description: prefetch_memory (base+offset)

PREF adds the 16-bit signed *offset* to the contents of GPR *base* to form an effective byte address. It advises that data at the effective address may be used in the near future.

If the hint field is 00000_2, this instruction prefetches a block of data from main memory into cache.

PREF is an advisory instruction. It may change the performance of the program. For all hint values and all effective addresses, it neither changes architecturally-visible state nor alters the meaning of the program.

PREF does not cause addressing-related exceptions. If it raises an exception condition, the exception conditions ignored. If an addressing-related exception condition is raised and ignored, no data will be prefetched, Even if no data is prefetched in such a case, some action that is not architecturally-visible, such as writeback of a dirty cache line, might take place.

PREF will never generate a memory operation for a location with an uncached memory access type.

The defined *hint* values are shown in the table below. The C790 only supports *hint* = 0. The *hint* table may be extended in future implementations.

Values of hint field for prefetch instruction

Value	Name	Data use and desired prefetch action
0	load	Data is expected to be loaded (not modified). Fetch data as if for a load.
1-31	(Reserved)	(Reserved)

Restrictions:

None

Operation:

vAddr ← sign_extend (offset) + GPR [base]$_{31..0}$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
Prefetch (uncached, pAddr, vAddr, DATA, hint)

Exceptions:

None

Programming Notes:

Prefetch can not prefetch data from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch on C790 may not prefetch data when there is outstanding bus read process due to a data cache miss, an uncached load or a miss on the uncached accelerated buffer.

Prefetch does not cause addressing exceptions. It will not cause an exception to prefetch using an address pointer value before the validity of a pointer determined.

Implementation Notes:

A reserved *hint* field value causes a default prefetch action, the load *hint*.

SB Store Byte SB

31–26	25–21	20–16	15–0
SB 101000	base	rt	offset
6	5	5	16

MIPS I

Format: SB rt, offset (base)

Purpose: To store a byte to memory.

Description: memory [base + offset] ← rt

The least-significant 8-bit byte of GPR *rt* is stored in memory at the location specified by the effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + GPR [base] $_{31..0}$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr$_{(PSIZE-1)..4}$ || (pAddr$_{3..0}$ xor BigEndian^4)
byte ← vAddr$_{3..0}$ xor BigEndian^4
dataquad ← GPR [rt]$_{(127-8*byte)..0}$ || 0^{8*byte}
StoreMemory (uncached, BYTE, dataquad, pAddr, vAddr, DATA)

Exceptions:

TLB Refill
TLB Invalid
TLB Modified
Address Error

Programming Notes:

None

SD Store Doubleword SD

31 26	25 21	20 16	15 0
SD 111111	base	rt	offset
6	5	5	16

MIPS III

Format: SD rt, offset (base)

Purpose: To store a doubleword to memory.

Description: memory [base + offset] ← rt

The 64-bit doubleword in GPR *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If any of the three least-significant bits of the effective address are non-zero, an Address Error exception occurs.

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + GPR $[base]_{31..0}$
if ($vAddr_{2..0}$) ≠ 0^3 then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← $pAddr_{(PSIZE-1)..4}$ || ($pAddr_{3..0}$ xor (BigEndian || 0^3))
byte ← $vAddr_{3..0}$ || (BigEndian || 0^3)
dataquad ← GPR $[rt]_{(127-8*byte)..0}$ || 0^{8*byte}
StoreMemory (uncached, DOUBLEWORD, dataquad, pAddr, vAddr, DATA)

Exceptions:

TLB Refill
TLB Invalid
TLB Modified
Address Error

Programming Notes:

None

SDL — Store Doubleword Left — SDL

31–26	25–21	20–16	15–0
SDL 101100	base	rt	offset
6	5	5	16

MIPS III

Format: SDL rt, offset (base)

Purpose: To store the more-significant part of a doubleword to an unaligned memory address.

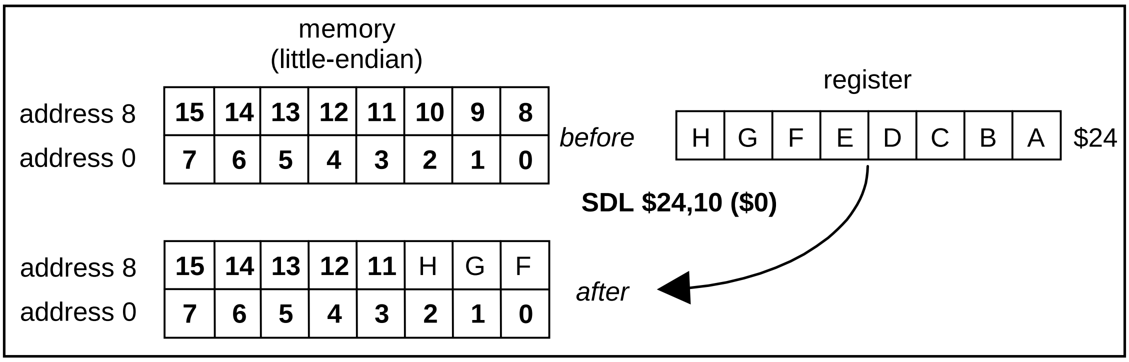
Description: memory [base + offset] ← rt

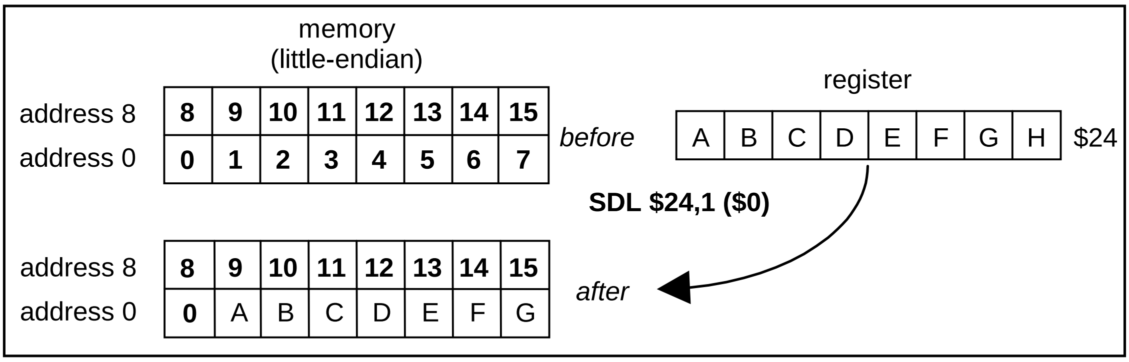
Paired SDL and SDR instructions are used to store a doubleword from a register into eight consecutive bytes in memory starting at an arbitrary byte address. SDL stores the left (most-significant) bytes and SDR stores the right (least-significant) bytes.

The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address of the most-significant byte of the contiguous doubleword in memory. It alters only the doubleword in memory which contains that byte. From one to eight bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the most-significant byte of the register and copies it to the specified byte in memory; then it copies bytes from register to memory until it reaches the low-order byte of the word in memory.

No address exceptions due to alignment are possible.





Restrictions:

None

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + GPR [base]$_{31..0}$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr$_{(PSIZE-1)..4}$ || (pAddr$_{3..0}$ xor BigEndian^{4})
If (BigEndian = 0) then
  pAddr ← pAddr$_{(PSIZE-1)..3}$ || 0^{3}
endif
byte ← 0 || (vAddr$_{2..0}$ xor BigEndian^{3})
if (vAddr$_{3}$ xor BigEndian = 0) then
  dataquad ← 0^{64} || $0^{(56-8*byte)}$ || GPR [rt]$_{63..(56-8*byte)}$
else
  dataquad ← $0^{(56-8*byte)}$ || GPR [rt]$_{63..(56-8*byte)}$ || 0^{64}
endif
StoreMemory (uncached, byte, dataquad, pAddr, vAddr, DATA)

Given a doubleword in a register and a doubleword in memory, the operation of SDL is as follows:

SDL

	MSB 63							0 LSB
Register	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**

Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Memory	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x

$vAddr_{3..0}$	Little-endian byte ordering (BigEndianCPU = 1)																		
	Destination memory contents after instruction(shaded is unchanged) (127------------------------------------64 63---0)																Type	offset	
																		LEM	BEM
0	I	j	k	l	m	n	o	p	q	r	s	t	u	v	w	**A**	0	8	15
1	I	j	k	l	m	n	o	p	q	r	s	t	u	v	**A**	**B**	1	8	14
2	I	j	k	l	m	n	o	p	q	r	s	t	u	**A**	**B**	**C**	2	8	13
3	I	j	k	l	m	n	o	p	q	r	s	t	**A**	**B**	**C**	**D**	3	8	12
4	I	j	k	l	m	n	o	p	q	r	s	**A**	**B**	**C**	**D**	**E**	4	8	11
5	I	j	k	l	m	n	o	p	q	r	**A**	**B**	**C**	**D**	**E**	**F**	5	8	10
6	I	j	k	l	m	n	o	p	q	**A**	**B**	**C**	**D**	**E**	**F**	**G**	6	8	9
7	I	j	k	l	m	n	o	p	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	7	8	8
8	I	j	k	l	m	n	o	**A**	q	r	s	t	u	v	w	x	8	0	7
9	I	j	k	l	m	n	**A**	**B**	q	r	s	t	u	v	w	x	9	0	6
10	I	j	k	l	m	**A**	**B**	**C**	q	r	s	t	u	v	w	x	10	0	5
11	I	j	k	l	**A**	**B**	**C**	**D**	q	r	s	t	u	v	w	x	11	0	4
12	I	j	k	**A**	**B**	**C**	**D**	**E**	q	r	s	t	u	v	w	x	12	0	3
13	I	j	**A**	**B**	**C**	**D**	**E**	**F**	q	r	s	t	u	v	w	x	13	0	2
14	I	**A**	**B**	**C**	**D**	**E**	**F**	**G**	q	r	s	t	u	v	w	x	14	0	1
15	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	q	r	s	t	u	v	w	x	15	0	0

SDL

	MSB 63							0 LSB
Register	A	B	C	D	E	F	G	H

Big-endian	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Memory	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x
Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

$vAddr_{3..0}$	Big-endian byte ordering (BigEndianCPU = 0)																		
	Destination memory contents after instruction(shaded is unchanged)																Type	offset	
	(127-----64							63								-----0)		LEM	BEM
0	A	B	C	D	E	F	G	H	q	r	s	t	u	v	w	x	15	0	0
1	i	A	B	C	D	E	F	G	q	r	s	t	u	v	w	x	14	0	1
2	i	j	A	B	C	D	E	F	q	r	s	t	u	v	w	x	13	0	2
3	i	j	k	A	B	C	D	E	q	r	s	t	u	v	w	x	12	0	3
4	i	j	k	l	A	B	C	D	q	r	s	t	u	v	w	x	11	0	4
5	i	j	k	l	m	A	B	C	q	r	s	t	u	v	w	x	10	0	5
6	i	j	k	l	m	n	A	B	q	r	s	t	u	v	w	x	9	0	6
7	i	j	k	l	m	n	o	A	q	r	s	t	u	v	w	x	8	0	7
8	i	j	k	l	m	n	o	p	A	B	C	D	E	F	G	H	7	0	8
9	i	j	k	l	m	n	o	p	q	A	B	C	D	E	F	G	6	0	9
10	i	j	k	l	m	n	o	p	q	r	A	B	C	D	E	F	5	0	10
11	i	j	k	l	m	n	o	p	q	r	s	A	B	C	D	E	4	0	11
12	i	j	k	l	m	n	o	p	q	r	s	t	A	B	C	D	3	0	12
13	i	j	k	l	m	n	o	p	q	r	s	t	u	A	B	C	2	0	13
14	i	j	k	l	m	n	o	p	q	r	s	t	u	v	A	B	1	0	14
15	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	A	0	0	15

LEM Little-endian memory (BigEndianMem = 0)
BEM BigEndianMem = 1
Type AccessLength sent to memory
Offset $pAddr_{3..0}$ sent to memory

Exceptions:

TLB Refill
TLB Invalid
TLB Modified
Address Error

Programming Notes:

None

SDR Store Doubleword Right SDR

31–26	25–21	20–16	15–0
SDR 101101	base	rt	offset
6	5	5	16

MIPS III

Format: SDR rt, offset (base)

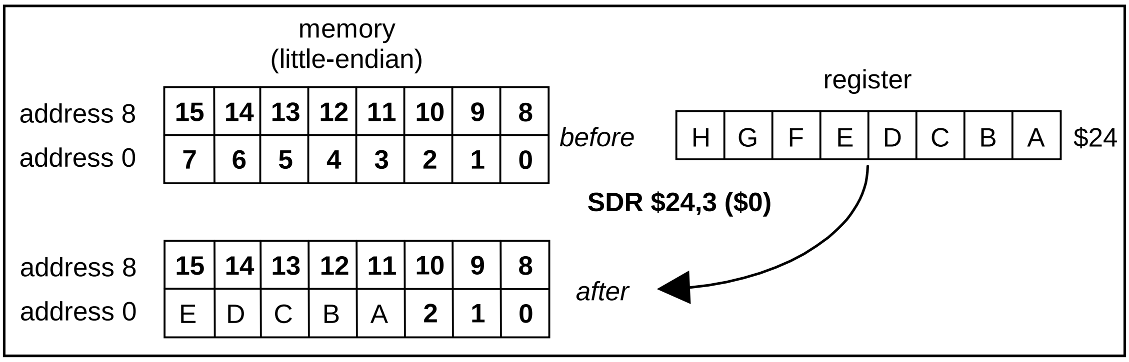
Purpose: To store the less-significant part of a doubleword to an unaligned memory address.

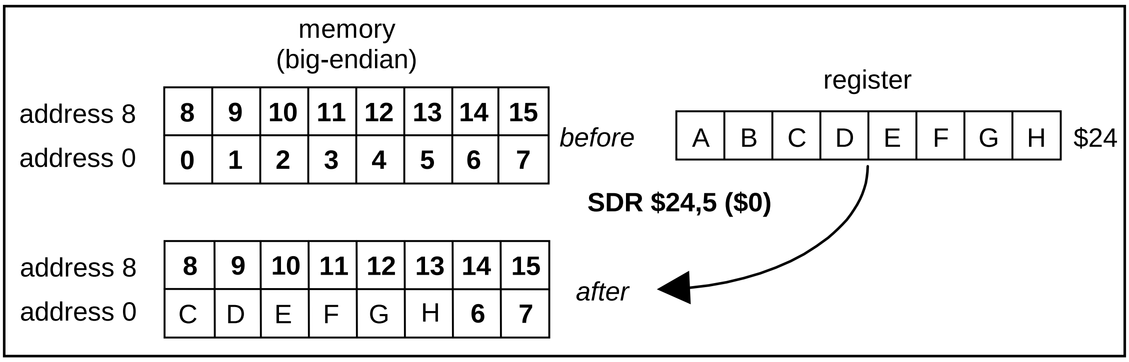
Description: memory [base + offset] ← rt

Paired SDL and SDR instructions are used to store a doubleword from a register into eight consecutive bytes in memory starting at an arbitrary byte address. SDL stores the left (most-significant) bytes and SDR stores the right (least-significant) bytes.

The SDR instruction adds its sign-extended 16-bit *offset* to the contents of GPR *base* to form an effective address which may specify an arbitrary byte. It alters only the doubleword in memory which contains that byte. From one to eight bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the least-significant (rightmost) byte of the register and copies it to the specified byte in memory; then it copies bytes from register to memory until it reaches the high-order byte of the word in memory. No address exceptions due to alignment are possible.





Restrictions:

None

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + GPR [base] $_{31..0}$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr$_{(PSIZE-1)..4}$ || (pAddr$_{3..0}$ xor BigEndian^4)
If (BigEndian = 0) then
  pAddr ← pAddr$_{(PSIZE-31)..3}$ || 0^3
endif
byte ← vAddr$_{2..0}$ xor BigEndian^4
if(vAddr$_3$ xor BigEndian = 0) then
  dataquad ← 0^{64} || GPR [rt]$_{(63-8*byte)..0}$ || 0^{8*byte}
else
  dataquad ← GPR [rt]$_{(63-8*byte)..0}$ || 0^{8*byte} || 0^{64}
endif
StoreMemory (uncached, DOUBLEWORD-byte, dataquad, pAddr, vAddr, DATA)

Given a doubleword in a register and a doubleword in memory, the operation of SDR is as follows:

SDR

	MSB 63							0 LSB
Register	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**

Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Memory	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x

vAddr$_{3..0}$	Little-endian byte ordering (BigEndianCPU = 0)																		
	Destination memory contents after instruction(shaded is unchanged)																Type	offset	
	(127---------------------------------------64 63--0)																	LEM	BEM
0	i	j	k	l	m	n	o	p	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	7	0	0
1	i	j	k	l	m	n	o	p	**B**	**C**	**D**	**E**	**F**	**G**	**H**	x	6	1	0
2	i	j	k	l	m	n	o	p	**C**	**D**	**E**	**F**	**G**	**H**	w	x	5	2	0
3	i	j	k	l	m	n	o	p	**D**	**E**	**F**	**G**	**H**	v	w	x	4	3	0
4	i	j	k	l	m	n	o	p	**E**	**F**	**G**	**H**	u	v	w	x	3	4	0
5	i	j	k	l	m	n	o	p	**F**	**G**	**H**	t	u	v	w	x	2	5	0
6	i	j	k	l	m	n	o	p	**G**	**H**	s	t	u	v	w	x	1	6	0
7	i	j	k	l	m	n	o	p	**H**	r	s	t	u	v	w	x	0	7	0
8	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	q	r	s	t	u	v	w	x	7	8	0
9	**B**	**C**	**D**	**E**	**F**	**G**	**H**	p	q	r	s	t	u	v	w	x	6	9	0
10	**C**	**D**	**E**	**F**	**G**	**H**	o	p	q	r	s	t	u	v	w	x	5	10	0
11	**D**	**E**	**F**	**G**	**H**	n	o	p	q	r	s	t	u	v	w	x	4	11	0
12	**E**	**F**	**G**	**H**	m	n	o	p	q	r	s	t	u	v	w	x	3	12	0
13	**F**	**G**	**H**	l	m	n	o	p	q	r	s	t	u	v	w	x	2	13	0
14	**G**	**H**	k	l	m	n	o	p	q	r	s	t	u	v	w	x	1	14	0
15	**H**	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	0	15	0

SDR

	MSB 63							0 LSB
Register	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**

Big-endian	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Memory	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x
Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

vAddr3..0	Big-endian byte ordering (BigEndianCPU = 0)																		
	Destination memory contents after instruction(shaded is unchanged) (127---------------------------------------64 63---0)																Type	offset	
																		LEM	BEM
0	**H**	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	0	15	0
1	**G**	**H**	k	l	m	n	o	p	q	r	s	t	u	v	w	x	1	14	0
2	**F**	**G**	**H**	l	m	n	o	p	q	r	s	t	u	v	w	x	2	13	0
3	**E**	**F**	**G**	**H**	m	n	o	p	q	r	s	t	u	v	w	x	3	12	0
4	**D**	**E**	**F**	**G**	**H**	n	o	p	q	r	s	t	u	v	w	x	4	11	0
5	**C**	**D**	**E**	**F**	**G**	**H**	o	p	q	r	s	t	u	v	w	x	5	10	0
6	**B**	**C**	**D**	**E**	**F**	**G**	**H**	p	q	r	s	t	u	v	w	x	6	9	0
7	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	q	r	s	t	u	v	w	x	7	8	0
8	i	j	k	l	m	n	o	p	**H**	r	s	t	u	v	w	x	0	7	0
9	i	j	k	l	m	n	o	p	**G**	**H**	s	t	u	v	w	x	1	6	0
10	i	j	k	l	m	n	o	p	**F**	**G**	**H**	t	u	v	w	x	2	5	0
11	i	j	k	l	m	n	o	p	**E**	**F**	**G**	**H**	u	v	w	x	3	4	0
12	i	j	k	l	m	n	o	p	**D**	**E**	**F**	**G**	**H**	v	w	x	4	3	0
13	i	j	k	l	m	n	o	p	**C**	**D**	**E**	**F**	**G**	**H**	w	x	5	2	0
14	i	j	k	l	m	n	o	p	**B**	**C**	**D**	**E**	**F**	**G**	**H**	x	6	1	0
15	i	j	k	l	m	n	o	p	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	7	0	0

LEM Little-endian memory (BigEndianMem = 0)
BEM BigEndianMem = 1
Type AccessLength sent to memory
Offset pAddr3..0 sent to memory

Exceptions:

TLB Refill
TLB Invalid
TLB Modified
Address Error

Programming Notes:

None

SH Store Halfword SH

31 26	25 21	20 16	15 0
SH 101001	base	rt	offset
6	5	5	16

MIPS I

Format: SH rt, offset (base)

Purpose: To store a halfword to memory.

Description: memory [base + offset] ← rt

The least-significant 16-bit halfword if register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + GPR $[base]_{31..0}$
if ($vAddr_0$) ≠ 0 then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← $pAddr_{(PSIZE-1)..4}$ || ($pAddr_{3..0}$ xor ($BigEndian^3$ || 0))
byte ← $vAddr_{3..0}$ xor ($BigEndian^3$ || 0)
dataquad ← GPR $[rt]_{(127-8*byte)..0}$ || 0^{8*byte}
StoreMemory (uncached, HALFWORD, dataquad, pAddr, vAddr, DATA)

Exceptions:

TLB Refill
TLB Invalid
TLB Modified
Address Error

Programming Notes:

None

SLL

Shift Word Left Logical

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	0 00000	rt	rd	sa	SLL 000000
6	5	5	5	5	6

MIPS I

Format: SLL rd, rt, sa

Purpose: To left shift a word by a fixed number of bits.

Description: rd ← rt << sa

The contents of the low-order 32-bit word of GPR *rt* are shifted left, inserting zeroes into the emptied bits; the word result is placed in GPR *rd*. The bit shift count is specified by *sa*. The result word is sign-extended.

Restrictions:

None

Operation:

s ← sa
temp ← GPR $[rt]_{(31-s)..0}$ || 0^s
GPR $[rd]_{63..0}$ ← sign_extend ($temp_{31..0}$)

Exceptions:

None

Programming Notes:

Unlike nearly all other word operations the input operand does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign extended into a 64-bit destination register; this instruction with a zero shift amount truncates a 64-bit value to 32 bits and sign extends it and stores it in the destination register.

SLLV

Shift Word Left Logical Variable

SLLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SLLV 000100
6	5	5	5	5	6

MIPS I

Format: SLLV rd, rt, rs

Purpose: To left shift a word by a variable number of bits.

Description: rd ← rt << rs

The contents of the low-order 32-bit word of GPR *rt* are shifted left, inserting zeroes into the emptied bits; the result word is placed in GPR *rd*. The bit shift count is specified by the low-order five bits of GPR *rs*. The result word is sign-extended.

Restrictions:

None

Operation:

s ← GP $[rs]_{4..0}$
temp ← GPR $[rt]_{(31-s)..0}$ || 0^s
GPR $[rd]_{63..0}$ ← sign_extend ($temp_{31..0}$)

Exceptions:

None

Programming Notes:

None

SLT Set on Less Than SLT

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SLT 101010
6	5	5	5	5	6

MIPS I

Format: SLT rd, rs, rt

Purpose: To record the result of a less-than comparison.

Description: rd ← (rs < rt)

Compare the contents of GPR *rs* and GPR *rt* as signed integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt* the result is 1 (true), otherwise 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

if GPR $[rs]_{63..0}$ < GPR $[rt]_{63..0}$ then
    GPR $[rd]_{63..0} \leftarrow 0^{GPRLEN-1} \| 1$
else
    GPR $[rd]_{63..0} \leftarrow 0^{GPRLEN}$
endif

Exceptions:

None

Programming Notes:

None

SLTI Set on Less Than Immediate SLTI

31 26	25 21	20 16	15 0
SLTI 001010	rs	rt	immediate
6	5	5	16

MIPS I

Format: SLTI rt, rs, immediate

Purpose: To record the result of a less-than comparison with a constant.

Description: rt ← (rs < immediate)

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate* the result is 1 (true), otherwise 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

```
if GPR [rs] 63..0 < sign_extend (immediate) then
        GPR [rd] 63..0 ← 0^(GPRLEN-1) || 1
else
        GPR [rd] 63..0 ← 0^GPRLEN
endif
```

Exceptions:

None

Programming Notes:

None

SLTIU Set on Less Than Immediate Unsigned SLTIU

31 26	25 21	20 16	15 0
SLTIU 001011	rs	rt	immediate
6	5	5	16

MIPS I

Format: SLTIU rt, rs, immediate

Purpose: To record the result of an unsigned less-than comparison with a constant.

Description: rt ← (rs < immediate)

Compare the contents of GPR *rs* and the sign-extended 16-bit *immediate* as unsigned integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate* the result is 1 (true), otherwise 0 (false).

Because the 16-bit *immediate* is sign-extended before comparison, the instruction is able to represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

if (0 ‖ GPR [rs] $_{63..0}$) < (0 ‖ sign_extend (immediate)) then
    GPR [rd] $_{63..0}$ ← $0^{GPRLEN-1}$ ‖ 1
else
    GPR [rd] $_{63..0}$ ← 0^{GPRLEN}
endif

Exceptions:

None

Programming Notes:

None

SLTU

Set on Less Than Unsigned

SLTU

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	SLTU 101011
6	5	5	5	5	6

MIPS I

Format: SLTU rd, rs, rt

Purpose: To record the result of an unsigned less-than comparison.

Description: rd ← (rs < rt)

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt* the result is 1 (true), otherwise 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

Operation:

if (0 || GPR [rs] $_{63..0}$) < (0 || GPR [rt] $_{63..0}$) then
    GPR [rd] $_{63..0}$ ← $0^{GPRLEN-1}$ || 1
else
    GPR [rd] $_{63..0}$ ← 0^{GPRLEN}
endif

Exceptions:

None

Programming Notes:

None

SRA Shift Word Right Arithmetic SRA

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SRA 000011
6	5	5	5	5	6

MIPS I

Format: SRA rd, rt sa

Purpose: To arithmetic right shift a word by a fixed number of bits.

Description: rd ← rt >> sa (arithmetic)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit shift count is specified by *sa*. The result word is sign-extended.

Restrictions:

If GPR *rt* does not contain a sign-extended 32-bit value (bit 63..31 equal) then the result of the operation is undefined.

Operation:

if (NotWordValue (GPR [rt]$_{63..0}$)) then UndefinedResult () endif
s ← sa
temp ← (GPR [rt]$_{31}$)^{s} || GPR [rt]$_{31..s}$
GPR [rd]$_{63..0}$ ← sign_extend (temp$_{31..0}$)

Exceptions:

None

Programming Notes:

None

SRAV Shift Word Right Arithmetic Variable SRAV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SRAV 000111
6	5	5	5	5	6

MIPS I

Format: SRAV rd, rt, rs

Purpose: To arithmetic right shift a word by a variable number of bits.

Description: rd ← rt >> rs (arithmetic)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit shift count is specified by the low-order five bits of GPR *rs*. The result word is sign-extended.

Restrictions:

If GPR *rt* does not contain a sign-extended 32-bit value (bit 63..31 equal) then the result of the operation is undefined.

Operation:

if (NotWordValue (GPR [rt] $_{63..0}$)) then UndefinedResult () endif
s ← GPR [rs]$_{4..0}$
temp ← (GPR [rt]$_{31}$)^{s} || GPR [rt]$_{31..s}$
GPR [rd] $_{63..0}$ ← sign_extend (temp$_{31..0}$)

Exceptions:

None

Programming Notes:

None

SRL Shift Word Right Logical SRL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SRL 000010
6	5	5	5	5	6

MIPS I

Format: SRL rd, rt, sa

Purpose: To logical right shift a word by a fixed number of bits.

Description: rd ← rt >> sa (logical)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit shift count is specified by *sa*. The result word is sign-extended.

Restrictions:

If GPR *rt* does not contain a sign-extended 32-bit value (bit 63..31 equal) then the result of the operation is undefined.

Operation:

if (NotWordValue (GPR [rt] $_{63..0}$)) then UndefinedResult () endif
s ← sa
temp ← 0^s || GPR [rt]$_{31..s}$
GPR [rd] $_{63..0}$ ← sign_extend(temp$_{31..0}$)

Exceptions:

None

Programming Notes:

None

SRLV Shift Word Right Logical Variable SRLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SRLV 000110
6	5	5	5	5	6

MIPS I

Format: SRLV rd, rt, rs

Purpose: To logical right shift a word by a variable number of bits.

Descriptions: rd ← rt >> rs (logical)

The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit shift count is specified by the low-order five bits of GPR *rs*. The result word is sign-extended.

Restrictions:

If GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal) then the result of the operation is undefined.

Operation:

if (NotWordValue (GPR[rt] $_{63..0}$)) then UndefinedResult () endif
s ← GPR [rs]$_{4..0}$
temp ← 0^s || GPR [rt]$_{31..s}$
GPR [rd]$_{63..0}$ ← sign_extend (temp$_{31..0}$)

Exceptions:

None

Programming Notes:

None

SUB Subtract Word SUB

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SUB 100010
6	5	5	5	5	6

MIPS I

Format: SUB rd, rs, rt

Purpose: To subtract 32-bit integers. If overflow occurs, then trap.

Description: rd ← rs - rt

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* to produce a 32-bit result. If the subtraction results in 32-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

```
if (NotWordValue (GPR[rs] 63..0) or NotWordValue (GPR[rt] 63..0)) then UndefinedResult () endif
temp ← GPR [rs] 63..0 - GPR [rt] 63..0
if (32_bit_arithmetic_overflow) then
      SignalException (IntegerOverflow)
else
      GPR [rd] 63..0 ← sign_extend (temp31..0)
endif
```

Exceptions:

Integer Overflow

Programming Notes:

SUBU performs the same arithmetic operation but, does not trap on overflow.

SUBU — Subtract Unsigned Word — SUBU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SUBU 100011
6	5	5	5	5	6

MIPS I

Format: SUBU rd, rs, rt

Purpose: To subtract 32-bit integers.

Description: rd ← rs - rt

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rd*.

No integer overflow exception occurs under any circumstances.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

if (NotWordValue (GPR[rs] $_{63..0}$) or NotWordValue (GPR[rt] $_{63..0}$)) then UndefinedResult () endif
temp ← GPR [rs] $_{63..0}$ - GPR [rt] $_{63..0}$
GPR [rd] $_{63..0}$ ← sign_extend (temp $_{31..0}$)

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

SW Store Word SW

31 26	25 21	20 16	15 0
SW 101011	base	rt	offset
6	5	5	16

MIPS I

Format: SW rt, offset (base)

Purpose: To store a word to memory.

Description: memory [base + offset] ← rt

The least-significant 32-bit word of register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If either of the two least-significant bits of the address are non-zero, an Address Error exception occurs.

Operation: (128-bit bus)

vAddr ← sign_extend (offset) + GPR [base] $_{31..0}$
if (vAddr$_{1..0}$) ≠ 0^2 then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr$_{(PSIZE-1)..4}$ || (pAddr$_{3..0}$ xor (BigEndian^2 || 0^2))
byte ← vAddr$_{3..0}$ xor (BigEndian^2 || 0^2)
dataquad ← GPR [rt]$_{(127-8*byte)..0}$ || 0^{8*byte}
StoreMemory (uncached, WORD, dataquad, pAddr, vAddr, DATA)

Exceptions:

TLB Refill
TLB Invalid
TLB Modified
Address Error

Programming Notes:

None

SWL Store Word Left SWL

31 … 26	25 … 21	20 … 16	15 … 0
SWL 101010	base	rt	offset
6	5	5	16

MIPS I

Format: SWL rt, offset (base)

Purpose: To store the more-significant part of a word to an unaligned memory address.

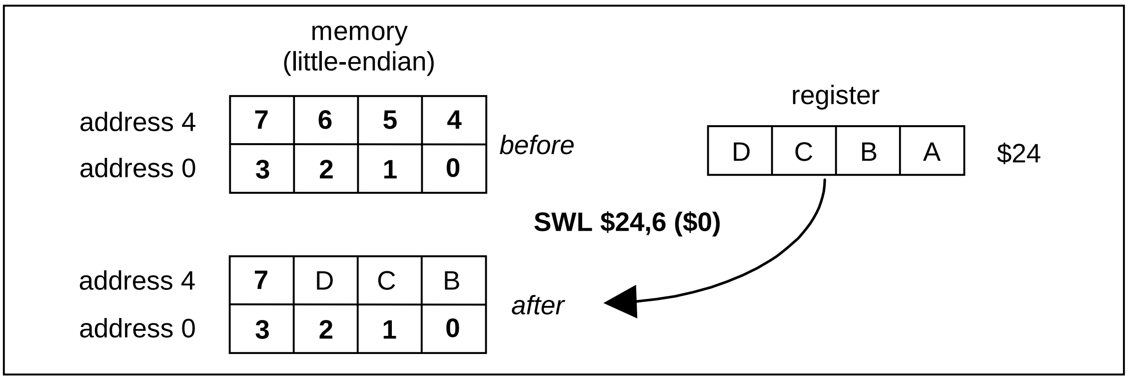
Description: memory [base + offset] ← rt

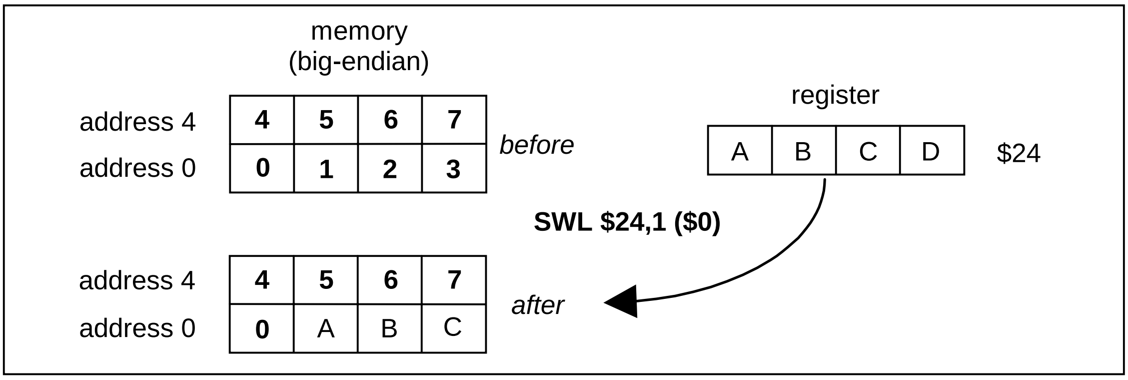
Paired SWL and SWR instructions are used to store a word from a register into four consecutive bytes in memory starting at an arbitrary byte address. SWL stores the left (most-significant) bytes and SWR stores the right (least-significant) bytes.

The SWL instruction adds its sign-extended 16-bit *offset* to the contents of GPR *base* to form an effective address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the most-significant byte of the register and copies it to the specified byte in memory; then it copies bytes from register to memory until it reaches the low-order byte of the word in memory.

No address exceptions due to alignment are possible.





Restrictions:

None

Operation:

vAddr ← sign_extend (offset) + GPR [base] $_{31..0}$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr$_{(PSIZE-1)..4}$ || (pAddr$_{3..0}$ xor BigEndian^4)
If (BigEndian = 0) then
    pAddr ← pAddr$_{(PSIZE-1)..2}$ || 0^2
endif
byte ← vAddr$_{1..0}$ xor BigEndian^2
if (vAddr$_{3..2}$ xor BigEndian^2) = 00_2 then
    dataquad ← 0^{96} || $0^{(24-8*byte)}$ || GPR[rt]$_{31..(24-8*byte)}$
elseif (vAddr$_{3..2}$ xor BigEndian^2) = 01_2 then
    dataquad ← 0^{64} || $0^{(24-8*byte)}$ || GPR [rt]$_{31..(24-8*byte)}$ || 0^{32}
elseif (vAddr$_{3..2}$ xor BigEndian^2) = 10_2 then
    dataquad ← 0^{32} || $0^{(24-8*byte)}$ || GPR [rt]$_{31..(24-8*byte)}$ || 0^{32}
elseif (vAddr$_{3..2}$ xor BigEndian^2) = 11_2 then
    dataquad ← $0^{(24-8*byte)}$ || GPR [rt]$_{31..(24-8*byte)}$ || 0^{64}
endif
StoreMemory (uncached, byte, dataquad, pAddr, vAddr, DATA)

Given a doubleword in a register and a doubleword in memory, the operation of SWL is as follows:

SWL

	MSB 63							0 LSB
Register	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**

Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Memory	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x

vAddr$_{3..0}$	Little-endian byte ordering (BigEndianCPU = 0)																		
	Destination memory contents after instruction(shaded is unchanged)																Type	offset	
	(127--------------------------------------64 63--0)																	LEM	BEM
0	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	**E**	0	0	15
1	i	j	k	l	m	n	o	p	q	r	s	t	u	v	**E**	**F**	1	0	14
2	i	j	k	l	m	n	o	p	q	r	s	t	u	**E**	**F**	**G**	2	0	13
3	i	j	k	l	m	n	o	p	q	r	s	t	**E**	**F**	**G**	**H**	3	0	12
4	i	j	k	l	m	n	o	p	q	r	s	**E**	u	v	w	x	0	4	11
5	i	j	k	l	m	n	o	p	q	r	**E**	**F**	u	v	w	x	1	4	10
6	i	j	k	l	m	n	o	p	q	**E**	**F**	**G**	u	v	w	x	2	4	9
7	i	j	k	l	m	n	o	p	**E**	**F**	**G**	**H**	u	v	w	x	3	4	8
8	i	j	k	l	m	n	o	**E**	q	r	s	t	u	v	w	x	0	8	7
9	i	j	k	l	m	n	**E**	**F**	q	r	s	t	u	v	w	x	1	8	6
10	i	j	k	l	m	**E**	**F**	**G**	q	r	s	t	u	v	w	x	2	8	5
11	i	j	k	l	**E**	**F**	**G**	**H**	q	r	s	t	u	v	w	x	3	8	4
12	i	j	k	**E**	m	n	o	p	q	r	s	t	u	v	w	x	0	12	3
13	i	j	**E**	**F**	m	n	o	p	q	r	s	t	u	v	w	x	1	12	2
14	i	**E**	**F**	**G**	m	n	o	p	q	r	s	t	u	v	w	x	2	12	1
15	**E**	**F**	**G**	**H**	m	n	o	p	q	r	s	t	u	v	w	x	3	12	0

SWL

		MSB 63							0 LSB
Register		**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**

Big-endian	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Memory	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x
Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

vAddr$_{3..0}$	Big-endian byte ordering (BigEndianCPU = 1)																		
	Destination memory contents after instruction(shaded is unchanged) (127---------------------------------------64 63--0)																Type	offset	
																		LEM	BEM
0	**E**	**F**	**G**	**H**	m	n	o	p	q	r	s	t	u	v	w	x	3	12	0
1	i	**E**	**G**	**H**	m	n	o	p	q	r	s	t	u	v	w	x	2	12	1
2	i	j	**E**	**F**	m	n	o	p	q	r	s	t	u	v	w	x	1	12	2
3	i	j	k	**E**	m	n	o	p	q	r	s	t	u	v	w	x	0	12	3
4	i	j	k	l	**E**	**F**	**G**	**H**	q	r	s	t	u	v	w	x	3	8	4
5	i	j	k	l	m	**E**	**F**	**G**	q	r	s	t	u	v	w	x	2	8	5
6	i	j	k	l	m	n	**E**	**F**	q	r	s	t	u	v	w	x	1	8	6
7	i	j	k	l	m	n	o	**E**	q	r	s	t	u	v	w	x	0	8	7
8	i	j	k	l	m	n	o	p	**E**	**F**	**G**	**H**	u	v	w	x	3	4	8
9	i	j	k	l	m	n	o	p	q	**E**	**F**	**G**	u	v	w	x	2	4	9
10	i	j	k	l	m	n	o	p	q	r	**E**	**F**	u	v	w	x	1	4	10
11	i	j	k	l	m	n	o	p	q	r	s	**F**	u	v	w	x	0	4	11
12	i	j	k	l	m	n	o	p	q	r	s	t	**E**	**F**	**G**	**H**	3	0	12
13	i	j	k	l	m	n	o	p	q	r	s	t	u	**E**	**F**	**G**	2	0	13
14	i	j	k	l	m	n	o	p	q	r	s	t	u	v	**E**	**F**	1	0	14
15	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	**F**	0	0	15

LEM Little-endian memory (BigEndianMem = 0)
BEM BigEndianMem = 1
Type AccessLength sent to memory
Offset pAddr$_{3..0}$ sent to memory

Exceptions:

TLB Refill
TLB Invalid
TLB Modified
Address Error

Programming Notes:

None

SWR Store Word Right SWR

31–26	25–21	20–16	15–0
SWR 101110	base	rt	offset
6	5	5	16

MIPS I

Format: SWR rt, offset (base)

Purpose: To store the less-significant part of a word to an unaligned memory address.

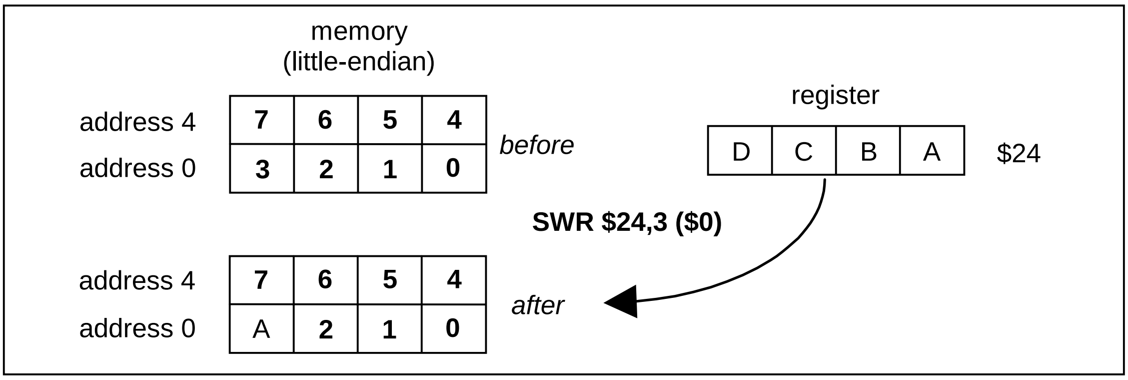
Description: memory [base + offset] ← rt

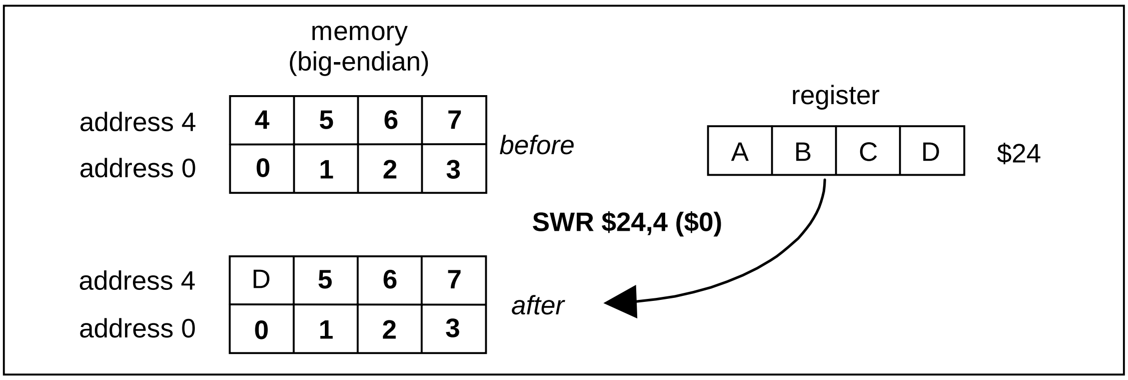
Paired SWL and SWR instructions are used to store a word from a register into four consecutive bytes in memory starting at an arbitrary byte address. SWL stores the left (most-significant) bytes and SWR stores the right (least-significant) bytes.

The SWR instruction adds its sign-extended 16-bit *offset* to the contents of GPR *base* to form an effective address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the least-significant (rightmost) byte of the register and copies it to the specified byte in memory; then copies bytes from register to memory until it reaches the high-order byte of the word in memory.

No address exceptions due to alignment are possible.





Restrictions:

None

Operation:

vAddr ← sign_extend (offset) + GPR [base] $_{31..0}$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr$_{(PSIZE-1)..4}$ || (pAddr$_{3..0}$ xor BigEndian^4)
If (BigEndian = 0) then
    pAddr ← pAddr$_{(PSIZE-1)..2}$ || 0^2
endif
byte ← vAddr$_{1..0}$ xor BigEndian^2
if (vAddr$_{3..2}$ xor BigEndian^2) = 00_2 then
    dataquad ← 0^{96} || GPR [rt] $_{(31-8*byte)..0}$ || 0^{8*byte}
else if (vAddr$_{3..2}$ xor BigEndian^2) = 01_2 then
    dataquad ← 0^{64} || GPR [rt] $_{(31-8*byte)..0}$ || 0^{8*byte} || 0^{32}
else if (vAddr$_{3..2}$ xor BigEndian^2) = 10_2 then
    dataquad ← 0^{32} || GPR [rt] $_{(31-8*byte)..0}$ || 0^{8*byte} || 0^{64}
else if (vAddr$_{3..2}$ xor BigEndian^2) = 11_2 then
    dataquad ←GPR [rt] $_{(31-8*byte)..0}$ || 0^{8*byte} || 0^{96}
endif
StoreMemory (uncached, WORD-byte, dataquad, pAddr, vAddr, DATA)

Given a doubleword in a register and a doubleword in memory, the operation of SWR is as follows:

SWR

	MSB 63						0 LSB
Register	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**

Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Memory	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x

vAddr$_{3..0}$	Little-endian byte ordering (BigEndianCPU = 0)																		
	Destination memory contents after instruction(shaded is unchanged)																Type	offset	
	(127--------------------------------------64 63--0)																	LEM	BEM
0	i	j	k	l	m	n	o	p	q	r	s	t	**E**	**F**	**G**	**H**	3	0	12
1	i	j	k	l	m	n	o	p	q	r	s	t	**F**	**G**	**H**	x	2	1	12
2	i	j	k	l	m	n	o	p	q	r	s	t	**G**	**H**	w	x	1	2	12
3	i	j	k	l	m	n	o	p	q	r	s	t	**H**	v	w	x	0	3	12
4	i	j	k	l	m	n	o	p	**E**	**F**	**G**	**H**	u	v	w	x	3	4	8
5	i	j	k	l	m	n	o	p	**F**	**G**	**H**	t	u	v	w	x	2	5	8
6	i	j	k	l	m	n	o	p	**G**	**H**	s	t	u	v	w	x	1	6	8
7	i	j	k	l	m	n	o	p	**H**	r	s	t	u	v	w	x	0	7	8
8	i	j	k	l	**E**	**F**	**G**	**H**	q	r	s	t	u	v	w	x	3	8	4
9	i	j	k	l	**F**	**G**	**H**	p	q	r	s	t	u	v	w	x	2	9	4
10	i	j	k	l	**G**	**H**	o	p	q	r	s	t	u	v	w	x	1	10	4
11	i	j	k	l	**H**	n	o	p	q	r	s	t	u	v	w	x	0	11	4
12	**E**	**F**	**G**	**H**	m	n	o	p	q	r	s	t	u	v	w	x	3	12	0
13	**F**	**G**	**H**	l	m	n	o	p	q	r	s	t	u	v	w	x	2	13	0
14	**G**	**H**	k	l	m	n	o	p	q	r	s	t	u	v	w	x	1	14	0
15	**H**	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	0	15	0

SWR

	MSB 63							0 LSB
Register	A	B	C	D	E	F	G	H

Big-endian	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Memory	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x
Little-endian	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

$vAddr_{3..0}$	Big-endian byte ordering (BigEndianCPU = 1)																		
	Destination memory contents after instruction(shaded is unchanged) (127--------64 63--------0)																Type	offset	
																		LEM	BEM
0	**H**	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	0	15	0
1	**G**	**H**	k	l	m	n	o	p	q	r	s	t	u	v	w	x	1	14	0
2	**F**	**G**	**H**	l	m	n	o	p	q	r	s	t	u	v	w	x	2	13	0
3	**E**	**F**	**G**	**H**	m	n	o	p	q	r	s	t	u	v	w	x	3	12	0
4	i	j	k	l	**H**	n	o	p	q	r	s	t	u	v	w	x	0	11	4
5	i	j	k	l	**G**	**H**	o	p	q	r	s	t	u	v	w	x	1	10	4
6	i	j	k	l	**F**	**G**	**H**	p	q	r	s	t	u	v	w	x	2	9	4
7	i	j	k	l	**E**	**F**	**G**	**H**	q	r	s	t	u	v	w	x	3	8	4
8	i	j	k	l	m	n	o	p	**H**	r	s	t	u	v	w	x	0	7	8
9	i	j	k	l	m	n	o	p	**G**	**H**	s	t	u	v	w	x	1	6	8
10	i	j	k	l	m	n	o	p	**F**	**G**	**H**	t	u	v	w	x	2	5	8
11	i	j	k	l	m	n	o	p	**E**	**F**	**G**	**H**	u	v	w	x	3	4	8
12	i	j	k	l	m	n	o	p	q	r	s	t	**H**	v	w	x	0	3	12
13	i	j	k	l	m	n	o	p	q	r	s	t	**G**	**H**	w	x	1	2	12
14	i	j	k	l	m	n	o	p	q	r	s	t	**F**	**G**	**H**	x	2	1	12
15	i	j	k	l	m	n	o	p	q	r	s	t	**E**	**F**	**G**	**H**	3	0	12

LEM Little-endian memory (BigEndianMem = 0)
BEM BigEndianMem = 1
Type AccessLength sent to memory
Offset $pAddr_{3..0}$ sent to memory

Exceptions:

TLB Refill
TLB Invalid
TLB Modified
Address Error

Programming Notes:

None

SYNC.stype

Synchronize Shared Memory

31 26	25 11	10 6	5 0
SPECIAL 000000	0 000 0000 0000 0000	stype	SYNC 001111
6	15	5	6

MIPS II

Format: SYNC (stype = 0xxxx)

SYNC.L (stype = 0xxxx)

SYNC.P (stype = 1xxxx)

Purpose: To perform either a memory barrier operation or a pipeline barrier operation.

Description:

This instruction either interlocks the pipeline until all pending loads and stores are completed or all earlier issued instructions are completed.

In case of the SYNC or the SYNC.L instructions (memory barrier) all pending loads and stores are retired. Loads are retired when the destination register is written. Stores are retired when the stored data (in store buffers or write buffers) is either stored in the data cache, or sent on the processor bus and SYSDACK* has been asserted. All uncached accelerated data gathering operation is terminated. The uncached accelerated buffer is invalidated. All bus read processes due to load/store/pref/cache instructions are completed. All pending bus write processes in the write back buffer are completed.

In case of the SYNC.P instruction (pipeline barrier) all instructions prior to the barrier are completed before the instructions following the barrier operation are fetched. Note that the barrier operation does not wait for any instruction which was issued prior to the barrier operation but not retired (e.g., multiply, divide, multicycle COP1 operations or a pending load which were issued prior to the barrier operation).

Operation:

SyncOperation (stype)

Exceptions:

None

Programming Notes:

The SYNC instruction (SYNC.P or SYNC.L) is not allowed in the branch delay slot of instructions which have branch delay slots.

SYSCALL

System Call

SYSCALL

31–26	25–6	5–0
SPECIAL 000000	code	SYSCALL 001100
6	20	6

MIPS I

Format: SYSCALL

Purpose: To cause a System Call exception.

Description:

A system call exception occurs, immediately and unconditionally transferring control to the exception handler.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Restrictions:

None

Operation:

SignalException (SystemCall)

Exceptions:

System Call

Programming Notes:

None

TEQ — Trap if Equal — TEQ

31–26	25–21	20–16	15–6	5–0
SPECIAL 000000	rs	rt	code	TEQ 110100
6	5	5	10	6

MIPS II

Format: TEQ rs, rt

Purpose: To compare GPRs and do a conditional Trap.

Description: if (rs = rt) then Trap

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is equal to GPR *rt* then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

None

Operation:

```
if GPR[rs]63..0 = GPR[rt] 63..0 then
        SignalException (Trap)
endif
```

Exceptions:

Trap

Programming Notes:

None

TEQI

Trap if Equal Immediate

TEQI

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TEQI 01100	immediate
6	5	5	16

MIPS II

Format: TEQI rs, immediate

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs = immediate) then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integer; if GPR *rs* is equal to *immediate* then taken a Trap exception.

Restrictions:

None

Operation:

```
if GPR [rs] 63..0 = sign_extend (immediate) then
        SignalException (Trap)
endif
```

Exceptions:

Trap

Programming Notes:

None

TGE

Trap if Greater or Equal

TGE

31 … 26	25 … 21	20 … 16	15 … 6	5 … 0
SPECIAL 000000	rs	rt	code	TGE 110000
6	5	5	10	6

MIPS II

Format: TGE rs, rt

Purpose: To compare GPRs and do a conditional Trap.

Description: if (rs ≥ rt) then Trap

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is greater than or equal to GPR *rt* then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

None

Operation:

```
if GPR [rs] 63..0 ≥ GPR [rt] 63..0 then
        SignalException (Trap)
endif
```

Exceptions:

Trap

Programming Notes:

None

TGEI

Trap if Greater or Equal Immediate

TGEI

31–26	25–21	20–16	15–0
REGIMM 000001	rs	TGEI 01000	immediate
6	5	5	16

MIPS II

Format: TGEI rs, immediate

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs ≥ immediate) then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is greater than or equal to *immediate* then take a Trap exception.

Restrictions:

None

Operation:

```
if GPR [rs] 63..0 ≥ sign_extend (immediate) then
        SignalException (Trap)
endif
```

Exceptions:

Trap

Programming Notes:

None

TGEIU

Trap if Greater or Equal Immediate Unsigned

TGEIU

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TGEIU 01001	immediate
6	5	5	16

MIPS II

Format: TGEIU rs, immediate

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs ≥ immediate) then Trap

Compare the contents of GPR *rs* and the 16-bit sign-extended *immediate* as unsigned integers; if GPR *rs* is greater than or equal to *immediate* then take a Trap exception.

Because the 16-bit *immediate* is sign-extended before comparison, the instruction is able to represent the smallest or largest unsigned numbers. The representable values are at the minimum [0,32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

Restrictions:

None

Operation:

```
if (0 || GPR[rs] 63..0) ≥ (0 || sign_extend (immediate)) then
        SignalException (Trap)
endif
```

Exceptions:

Trap

Programming Notes:

None

TGEU Trap if Greater or Equal Unsigned TGEU

31–26	25–21	20–16	15–6	5–0
SPECIAL 000000	rs	rt	code	TGEU 110001
6	5	5	10	6

MIPS II

Format: TGEU rs, rt

Purpose: To compare GPRs and do a conditional Trap.

Description: if (rs ≥ rt) then Trap

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers; if GPR *rs* is greater than or equal to GPR *rt* then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

None

Operation:

```
if (0 || GPR[rs] 63..0)) ≥ (0 || GPR[rt] 63..0) then
      SignalException (Trap)
endif
```

Exceptions:

Trap

Programming Notes:

None

TLT Trap if Less Than TLT

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TLT 110010
6	5	5	10	6

MIPS II

Format: TLT rs, rt

Purpose: To compare GPRs and do a conditional Trap.

Description: if (rs < rt) then Trap

Compare the contents of GPR *rs* and GPR *rs* as signed integers; if GPR *rs* is less than GPR *rt* then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

None

Operation:

```
if GPR [rs] 63..0 < GPR [rt] 63..0 then
        SignalException (Trap)
endif
```

Exceptions:

Trap

Programming Notes:

None

TLTI

Trap if Less Than Immediate

31–26	25–21	20–16	15–0
REGIMM 000001	rs	TLTI 01010	immediate
6	5	5	16

MIPS II

Format: TLTI rs, immediate

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs < immediate) then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is less than *immediate* then take a Trap exception.

Restrictions:

None

Operation:

```
if GPR[rs] 63..0 < sign_extend (immediate) then
        SignalException (Trap)
endif
```

Exceptions:

Trap

Programming Notes:

None

TLTIU Trap if Less Than Immediate Unsigned TLTIU

31–26	25–21	20–16	15–0
REGIMM 000001	rs	TLTIU 01011	immediate
6	5	5	16

MIPS II

Format: TLTIU rs, immediate

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs < immediate) then Trap

Compare the contents of GPR *rs* and the 16-bit sign-extended *immediate* as unsigned integers; if GPR *rs* is less than *immediate* then take a Trap exception.

Because the 16-bit *immediate* is sign-extended before comparison, the instruction is able to represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

Restrictions:

None

Operation:

```
if (0 || GPR[rs] 63..0) < (0 || sign_extend (immediate)) then
        SignalException (Trap)
endif
```

Exceptions:

Trap

Programming Notes:

None

TLTU   Trap if Less Than Unsigned   TLTU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TLTU 110011
6	5	5	10	6

MIPS II

Format: TLTU rs, rt

Purpose: To compare GPRs and do a conditional Trap.

Description: if (rs < rt) then Trap

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers; if GPR *rs* is less than GPR *rt* then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

None

Operation:

if (0 || GPR[rs]$_{63..0}$) < (0 || GPR[rt]$_{63..0}$) then
    SignalException (Trap)
endif

Exceptions:

Trap

Programming Notes:

None

TNE Trap if Not Equal TNE

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TNE 110110
6	5	5	10	6

MIPS II

Format: TNE rs, rt

Purpose: To compare GPRs and do a conditional Trap.

Description: if (rs ≠ rt) then Trap

Compare the contents of GPR *rs* and GPR *rt* as signed integers; if GPR *rs* is not equal to GPR *rt* then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software must load the instruction word from memory.

Restrictions:

None

Operation:

```
if GPR[rs] 63..0 ≠ GPR[rt] 63..0 then
        SignalException (Trap)
endif
```

Exceptions:

Trap

Programming Notes:

None

TNEI — Trap if Not Equal Immediate — TNEI

31–26	25–21	20–16	15–0
REGIMM 000001	rs	TNEI 01110	immediate
6	5	5	16

MIPS II

Format: TNEI rs, immediate

Purpose: To compare a GPR to a constant and do a conditional Trap.

Description: if (rs ≠ immediate) then Trap

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; if GPR *rs* is not equal to *immediate* then take a Trap exception.

Restriction:

None

Operation:

```
if GPR[rs] 63..0 ≠ sign_extend (immediate) then
      SignalException (Trap)
endif
```

Exceptions:

Trap

Programming Notes:

None

XOR

Exclusive OR

XOR

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	XOR 100110
6	5	5	5	5	6

MIPS I

Format: XOR rd, rs, rt

Purpose: To do a bitwise logical EXCLUSIVE OR.

Description: rd ← rs XOR rt

Combine the contents of GPR *rs* and GPR *rt* in a bitwise logical exclusive OR operation and place the result into GPR *rd*.

Restrictions:

None

Operation:

$GPR[rd]_{63..0} \leftarrow GPR[rs]_{63..0}$ xor $GPR[rt]_{63..0}$

Exceptions:

None

Programming Notes:

None

XORI

Exclusive OR Immediate

XORI

31 26	25 21	20 16	15 0
XORI 001110	rs	rt	immediate
6	5	5	16

MIPS I

Format: XORI rt, rs, immediate

Purpose: To do a bitwise logical EXCLUSIVE OR with a constant.

Description: rt ← rs XOR immediate

Combine the contents of GPR *rs* and the 16-bit zero-extended *immediate* in a bitwise logical exclusive OR operation and place the result into GPR *rt*.

Restrictions:

None

Operation:

$GPR[rt]_{63..0} \leftarrow GPR[rs]_{63..0}$ xor zero_extend (immediate)

Exceptions:

None

Programming Notes:

None

A.5 CPU Instruction Encoding

The following table shows the OpCode encoding of CPU instructions for the MIPS IV architecture. This architecture level includes all MIPS I, MIPS II, MIPS III and some MIPS IV instructions. Even though the OpCodes for MTSAB, MTSAH, MFSA, MTSA, LQ, and SQ are shown in this OpCode table, these instructions are described in Appendix B since they are C790-specific instructions.

Coprocessor 0 (COP0 - System Control Processor), Coprocessor 1 (COP1 - Floating-point Processor) and C790 specific instructions are described in separate sections.

31 26	0
OpCode	

OpCode bits 28..26 — Instructions encoded by **OpCode** field

bits 31..29	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	***SPECIAL*** δ	***REGIMM*** δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1 001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2 010	***COP0*** α, λ	***COP1*** α, π	*	*	BEQL	BNEL	BLEZL	BGTZL
3 011	DADDI	DADDIU	LDL	LDR	***MMI*** δ, μ	*	LQ μ	SQ μ
4 100	LB	LH	LWL	LW	LBU	LHU	LWR	LWU
5 101	SB	SH	SWL	SW	SDL	SDR	SWR	CACHE
6 110	η	LWC1	η	PREF	η	LDC1	η	LD
7 111	η	SWC1	η	*	η	SDC1	η	SD

31 26		5 0
OpCode = SPECIAL		**function**

function bits 2..0 — Instructions encoded by **function** field when OpCode field = SPECIAL

bits 5..3	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1 001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2 010	MFHI	MTHI	MFLO	MTLO	DSLLV	*	DSRLV	DSRAV
3 011	MULT	MULTU	DIV	DIVU	η	η	η	η
4 100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5 101	MFSA μ	MTSA μ	SLT	SLTU	DADD	DADDU	DSUB	DSUBU
6 110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7 111	DSLL	*	DSRL	DSRA	DSLL32	*	DSRL32	DSRA32

31 26	20 16		0
OpCode = REGIMM		**rt**	

rt bits 18..16 Instructions encoded by **rt** field when OpCode field = REGIMM

bits 20..19	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 00	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
0 01	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2 10	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3 11	MTSAB μ	MTSAH μ	*	*	*	*	*	*

* This OpCode is reserved for future use. An attempt to execute it causes a Reserved Instruction exception.

η This OpCode is reserved for one of the following instructions which are currently not supported: DMULT, DMULTU, DDIV, DDIVU, LL, LLD, SC, SCD, LWC2, SWC2. An attempt to execute it causes a Reserved Instruction exception.

δ This OpCode indicates an instruction class. The instruction word must be further decoded by examining additional tables that show the values for another instruction field.

μ This OpCode indicates C790 specific instructions. It is included in the table because it uses a primary OpCode in the instruction encoding map.

α This OpCode is a coprocessor operation, not a CPU operation. If the processor state does not allow access to the specified coprocessor, the instruction causes a Coprocessor Unusable exception. It is included in the table because it uses a primary OpCode in the instruction encoding map.

λ This OpCode indicates the class of Coprocessor 0 (System Control Processor) instructions. If the processor state does not allow access to the coprocessor 0, the instruction causes a Coprocessor Unusable exception. Further encoding information for this instruction class is in the COP0 Instruction Encoding tables.

π This OpCode indicates the class of Coprocessor 1 (Floating-Point Processor) instructions. If the processor state does not allow access to the coprocessor 1, the instruction causes a Coprocessor Unusable exception. Further encoding information for this instruction class is in the COP1 Instruction Encoding tables.

B. C790-Specific Instruction Set Details

This appendix provides a detailed description of the operation of each C790-specific instruction. The C790's instruction set is extended from the original MIPS ISA in order to support embedded applications. There are three classes of C790-specific instructions:

- Three-operand Multiply and Multiply-Add instructions
- Multiply and Multiply-Add instructions for pipeline 1
- Multimedia instructions

B.1 Conventions Used in This Chapter

The *HI* and *LO* registers are 128 bits wide. Some instructions operate on either the lower or the upper doublewords of these registers, and there are also instructions which operate on the complete registers.

The following terminology is used for these registers.

- Strictly speaking, a reference to the least-significant doubleword of the *HI* and *LO* register should use the names *HI0* and *LO0*. However, to be consistent with existing MIPS terminology, these registers are just called *HI* and *LO*.
- Reference to the upper doublewords of the *HI* and *LO* registers is made by using the names *HI1* and *LO1*.
- Occasionally, based on context, the complete 128-bit registers are referred to as *HI* and *LO*.
- Any portion of these registers can use the names *HI* and *LO* with the appropriate bit width specifications. Thus *HI1* can be referred to as $HI_{127..64}$ and *LO1* can be referred to as $LO_{127..64}$, etc.

B.1.1 Instruction Description Notation and Functions

The *Operation* sections of the instruction descriptions describe the operation performed by each instruction using a high-level language notation, or pseudocode. Symbols, functions, and structures used in the *Operation* sections are described here.

B.1.2 Pseudocode Language Statement Execution

Each of the high-level language statements in an operation description is executed in sequential order (as modified by conditional and loop constructs).

B.1.3 Pseudocode Symbols

Special symbols used in the notation are described in Appendix A.

B.2 Definitions for Pseudocode Functions Used in Operation Descriptions

A variety of functions are used in the pseudocode descriptions to make the pseudocode more readable and also to abstract implementation-specific behavior. These functions are defined in Appendix A.

B.3 Summary of C790-Specific Instructions

B.3.1 Multiply and Multiply-Add Instructions

- **Three-Operand Multiply and Multiply-Add (4 instructions)**

MADD	Multiply/Add
MADDU	Multiply/Add Unsigned
MULT	Multiply (3-operand)
MULTU	Multiply Unsigned (3-operand)

- **Multiply Instructions for Pipeline 1 (10 instructions)**

MULT1	Multiply Pipeline 1
MULTU1	Multiply Unsigned Pipeline 1
DIV1	Divide Pipeline 1
DIVU1	Divide Unsigned Pipeline 1
MADD1	Multiply-Add Pipeline 1
MADDU1	Multiply-Add Unsigned Pipeline 1
MFHI1	Move From HI1 Register
MFLO1	Move From LO1 Register
MTHI1	Move To HI1 Register
MTLO1	Move To LO1 Register

B.3.2 Multimedia Instructions

- **Arithmetic (19 instructions)**

PADDB	Parallel Add Byte
PSUBB	Parallel Subtract Byte
PADDH	Parallel Add Halfword
PSUBH	Parallel Subtract Halfword
PADDW	Parallel Add Word
PSUBW	Parallel Subtract Word
PADSBH	Parallel Add/Subtract Halfword
PADDSB	Parallel Add with Signed Saturation Byte
PSUBSB	Parallel Subtract with Signed Saturation Byte
PADDSH	Parallel Add with Signed Saturation Halfword
PSUBSH	Parallel Subtract with Signed Saturation Halfword
PADDSW	Parallel Add with Signed Saturation Word
PSUBSW	Parallel Subtract with Signed Saturation Word
PADDUB	Parallel Add with Unsigned saturation Byte
PSUBUB	Parallel Subtract with Unsigned saturation Byte
PADDUH	Parallel Add with Unsigned saturation Halfword
PSUBUH	Parallel Subtract with Unsigned saturation Halfword
PADDUW	Parallel Add with Unsigned saturation Word
PSUBUW	Parallel Subtract with Unsigned saturation Word

- **Min/Max (4 instructions)**

PMAXH	Parallel Maximum Halfword
PMINH	Parallel Minimum Halfword
PMAXW	Parallel Maximum Word
PMINW	Parallel Minimum Word

- **Absolute (2 instructions)**

PABSH	Parallel Absolute Halfword
PABSW	Parallel Absolute Word

- **Logical (4 instructions)**

PAND	Parallel AND
POR	Parallel OR
PXOR	Parallel XOR
PNOR	Parallel NOR

- **Shift (9 instructions)**

PSLLH	Parallel Shift Left Logical Halfword
PSRLH	Parallel Shift Right Logical Halfword
PSRAH	Parallel Shift Right Arithmetic Halfword
PSLLW	Parallel Shift Left Logical Word
PSRLW	Parallel Shift Right Logical Word
PSRAW	Parallel Shift Right Arithmetic Word
PSLLVW	Parallel Shift Left Logical Variable Word
PSRLVW	Parallel Shift Right Logical Variable Word
PSRAVW	Parallel Shift Right Arithmetic Variable Word

- **Compare (6 instructions)**

PCGTB	Parallel Compare for Greater Than Byte
PCEQB	Parallel Compare for Equal Byte
PCGTH	Parallel Compare for Greater Than Halfword
PCEQH	Parallel Compare for Equal Halfword
PCGTW	Parallel Compare for Greater Than Word
PCEQW	Parallel Compare for Equal Word

- **LZC (1 instruction)**

PLZCW	Parallel Leading Zero or One Count Word

- **Quadword Load and Store (2 instructions)**

LQ	Load Quadword
SQ	Store Quadword

- **Multiply and Divide (19 instructions)**

PMULTW	Parallel Multiply Word
PMULTUW	Parallel Multiply Unsigned Word
PDIVW	Parallel Divide Word
PDIVUW	Parallel Divide Unsigned Word
PMADDW	Parallel Multiply-Add Word
PMADDUW	Parallel Multiply-Add Unsigned Word
PMSUBW	Parallel Multiply-Subtract Word
PMULTH	Parallel Multiply Halfword
PMADDH	Parallel Multiply-Add Halfword
PMSUBH	Parallel Multiply-Subtract Halfword
PHMADH	Parallel Horizontal Multiply-Add Halfword
PHMSBH	Parallel Horizontal Multiply-Subtract Halfword
PDIVBW	Parallel Divide Broadcast Word
PMFHI	Parallel Move From HI Register
PMFLO	Parallel Move From LO Register
PMTHI	Parallel Move To HI Register
PMTLO	Parallel Move To LO Register
PMFHL	Parallel Move From HI/LO Register
PMTHL	Parallel Move To HI/LO Register

- **Pack/Extend (11 instructions)**

PPAC5	Parallel Pack to 5 bits
PPACB	Parallel Pack to Byte
PPACH	Parallel Pack to Halfword
PPACW	Parallel Pack to Word
PEXT5	Parallel Extend Upper from 5 bits
PEXTUB	Parallel Extend Upper from Byte
PEXTLB	Parallel Extend Lower from Byte
PEXTUH	Parallel Extend Upper from Halfword
PEXTLH	Parallel Extend Lower from Halfword
PEXTUW	Parallel Extend Upper from Word
PEXTLW	Parallel Extend Lower from Word

- **Others (16 instructions)**

PCPYH	Parallel Copy Halfword
PCPYLD	Parallel Copy Lower Doubleword
PCPYUD	Parallel Copy Upper Doubleword
PREVH	Parallel Reverse Halfword
PINTH	Parallel Interleave Halfword
PINTEH	Parallel Interleave Even Halfword
PEXEH	Parallel Exchange Even Halfword
PEXCH	Parallel Exchange Center Halfword
PEXEW	Parallel Exchange Even Word
PEXCW	Parallel Exchange Center Word
QFSRV	Quadword Funnel Shift Right Variable
MFSA	Move from Shift Amount Register
MTSA	Move to Shift Amount Register
MTSAB	Move Byte Count to Shift Amount Register
MTSAH	Move Halfword Count to Shift Amount Register
PROT3W	Parallel Rotate 3 Words

B.4 Instruction Set Details

In the following sections, details are provided for each of the C790-specific instructions.

Exceptions that may occur due to the execution of each instruction are listed after the description of each instruction. Descriptions of the immediate cause and manner of handling exceptions are omitted from the instruction descriptions in this appendix.

DIV1 Divide Word Pipeline 1 DIV1

31 26	25 21	20 16	15 6	5 0
MMI 011100	rs	rt	0 0000000000	DIV1 011010
6	5	5	10	6

C790

Format: DIV1 rs, rt

Purpose: To divide 32-bit signed integers using pipeline 1.

Description: (LO1, HI1) ← rs / rt

The 32-bit value in GPR *rs* is divided by the 32-bit value in GPR *rt*, treating both operands as signed values. The 32-bit quotient is placed into special register *LO1* (= $LO_{127..64}$) and the 32-bit remainder is placed into special register *HI1* (= $HI_{127..64}$).

No arithmetic exception occurs under any circumstances.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation will be undefined.

If the divisor in GPR *rt* is zero, the arithmetic result value will be undefined.

Operation:

if (NotWordValue(GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif

$q \leftarrow GPR[rs]_{31..0}$ div $GPR[rt]_{31..0}$

$r \leftarrow GPR[rs]_{31..0}$ mod $GPR[rt]_{31..0}$

$LO_{127..64} \leftarrow (q_{31})^{32} \| q_{31..0}$

$HI_{127..64} \leftarrow (r_{31})^{32} \| r_{31..0}$

Supplementary Explanation:

Normally, when 0x80000000 (-2147483648) the signed minimum value is divided by 0xFFFFFFFF (-1), the operation will result in an overflow. However, in this instruction an overflow exception doesn't occur and the result will be as follows:

Quotient is 0x80000000 (-2147483648), and remainder is 0x00000000 (0).

This sign of the quotient and the remainder is based on the signs of the dividend and the divisor as shown in the table below:

Table B-1. Quotient and Remainder Signs

Dividend	Divisor	Quotient	Remainder
Positive	Positive	Positive	Positive
Positive	Negative	Negative	Positive
Negative	Positive	Negative	Negative
Negative	Negative	Positive	Negative

Exceptions:

None

Programming Notes:

In C790, the integer divide operation proceeds asynchronously and allows other CPU instructions to execute before it is retired. An attempt to read *LO1* or *HI1* registers before the results are written will cause an interlock until the results are ready. Out-of-order execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the divide so that other instructions can execute in parallel.

No arithmetic exception occurs under any circumstances. Divide-by-zero or overflow conditions should be detected by instructions preceding the divide instruction. If the divide is asynchronous then the zero-divisor check can execute in parallel with the divide. The action taken on either divide-by-zero or overflow is either a convention within the program itself or more typically, the system software; one possibility is to take a BREAK exception with a code field value to signal the problem to the system software.

As an example, the C programming language in a UNIX environment expects division by zero to either terminate the program or execute a program-specified signal handler. C does not expect overflow to cause any exceptional condition. If the C compiler uses a divide instruction, it also emits code to test for a zero divisor and execute a BREAK instruction to inform the operating system if one is detected.

DIVU1 Divide Unsigned Word Pipeline 1 DIVU1

31 26	25 21	20 16	15 6	5 0
MMI 011100	rs	rt	0 0000000000	DIVU1 011011
6	5	5	10	6

C790

Format: DIVU1 rs, rt

Purpose: To divide 32-bit unsigned integers using pipeline 1.

Description: (LO1, HI1) ← rs / rt

The 32-bit value in GPR *rs* is divided by the 32-bit value in GPR *rt*, treating both operands as unsigned values. The 32-bit quotient is placed into special register *LO1* (= $LO_{127..64}$) and the 32-bit remainder is placed into special register *HI1* (= $HI_{127..64}$).

No arithmetic exception occurs under any circumstances.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain zero-extended 32-bit values (bits 63..32 equal zero), then the result of the operation is undefined.

If the divisor in GPR *rt* is zero, the arithmetic result will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif

$q \leftarrow (0 \parallel GPR[rs]_{31..0})$ div $(0 \parallel GPR[rt]_{31..0})$

$r \leftarrow (0 \parallel GPR[rs]_{31..0})$ mod $(0 \parallel GPR[rt]_{31..0})$

$LO_{127..64} \leftarrow (q_{31})^{32} \parallel q_{31..0}$

$HI_{127..64} \leftarrow (r_{31})^{32} \parallel r_{31..0}$

Exceptions:

None

Programming Notes:

See the Programming Notes for the DIV1 instruction.

LQ Load Quadword LQ

31 — 26	25 — 21	20 — 16	15 — 0
LQ 011110	base	rt	offset
6	5	5	16

C790

Format: LQ rt, offset (base)

Purpose: To load a quadword from memory.

Description: rt ← memory [base + offset]

The contents of the 128-bit quadword at the memory location specified by the effective address are fetched and placed in the 128-bit GPR *rt*. The 16-bit signed offset is added to the contents of GPR base register to form the effective address. The least-significant four bits of the effective address are masked to zero (effectively creating an aligned address) before being used to access memory. No address exceptions due to alignment are possible.

Restriction:

The effective address doesn't have to be naturally aligned. The least significant 4 bits of the effective address are ignored.

Operations:

$vAddr \leftarrow$ sign_extend (offset) + GPR $[base]_{31..0}$

$vAddr_{3..0} = 0^4$

(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)

memquad ← LoadMemory (uncached, QUADWORD, pAddr, vAddr, DATA)

$GPR[rt]_{127..0} \leftarrow$ memquad

Exceptions:

TLB Refill
TLB Invalid
Address Error

MADD

Multiply-Add word

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	0 00000	MADD 000000
6	5	5	5	5	6

C790

Format: MADD rs, rt

MADD rd, rs, rt

Purpose: To multiply 32-bit signed integers and add.

Description: (rd, HI, LO) ← (HI, LO) + rs × rt

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as signed values, to produce a 64-bit multiply result. The 64-bit multiply result is added to the contents in special registers *HI* and *LO*. The low-order 32-bit word of the result is placed into special register *LO* and GPR *rd*, and the high-order 32-bit word of the result is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

If GPR *rd* is omitted in assembly language, 0 is used as the default value.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod ← $(HI_{31..0} \| LO_{31..0}) + GPR[rs]_{31..0} * GPR[rt]_{31..0}$
$LO_{63..0}$ ← $(prod_{31})^{32} \| prod_{31..0}$
$HI_{63..0}$ ← $(prod_{63})^{32} \| prod_{63..32}$
$GPR[rd]_{63..0}$ ← $(prod_{31})^{32} \| prod_{31..0}$

Exceptions:

None

Programming Notes:

In C790, the integer multiply accumulate operation proceeds asynchronously and allows other CPU instructions to execute before it is retired. An attempt to read *LO* or *HI* registers before the results are written will cause an interlock until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

MADD1 — Multiply-Add word Pipeline 1 — MADD1

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	0 00000	MADD1 100000
6	5	5	5	5	6

C790

Format: MADD1 rs, rt

MADD1 rd, rs, rt

Purpose: To multiply 32-bit signed integers and add in Pipeline 1.

Description: (rd, HI1, LO1) ← (HI1, LO1) + rs × rt

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as signed values, to produce a 64-bit multiply result. The 64-bit multiply result is added to the contents in special registers *HI1* (= $HI_{127..64}$) and *LO1* (= $LO_{127..64}$). The low-order 32-bit word of the result is placed into special register *LO1* and GPR *rd*, and the high-order 32-bit word of the result is placed into special register *HI1*.

No arithmetic exception occurs under any circumstances.

If GPR *rd* is omitted in assembly language, 0 is used as the default value.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation will be undefined.

Operation:

```
if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod            ← (HI95..64 || LO95..64) + GPR[rs]31..0 * GPR[rt]31..0
LO127..64       ← (prod 31)^32 || prod31..0
HI127..64       ← (prod 63)^32 || prod63..32
GPR[rd]63..0    ← (prod 31)^32 || prod31..0
```

Exceptions:

None

Programming Notes:

In the C790, the integer multiply accumulate operation proceeds asynchronously and allows other CPU instructions to execute before it is retired. An attempt to read *LO1* or *HI1* registers before the results are written will cause an interlock until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

MADDU Multiply-Add Unsigned word MADDU

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	0 00000	MADDU 000001
6	5	5	5	5	6

C790

Format: MADDU rs, rt

MADDU rd, rs, rt

Purpose: To multiply 32-bit unsigned integers and add.

Description: (rd, HI, LO) ← (HI, LO) + rs × rt

The 32-bit word value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as unsigned values, to produce a 64-bit multiply result. The 64-bit multiply result is added to the contents in special registers *HI* and *LO*. The low-order 32-bit word of the result is placed into special register *LO* and GPR *rd*, and the high-order 32-bit word of the result is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

If GPR *rd* is omitted in assembly language, 0 is used as the default value.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain zero-extended 32-bit values (bits 63..32 equal zero), then the result of the operation will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod ← ($HI_{31..0}$ || $LO_{31..0}$) + (0 || $GPR[rs]_{31..0}$) * (0 || $GPR[rt]_{31..0}$)
$LO_{63..0}$ ← $(prod_{31})^{32}$ || $prod_{31..0}$
$HI_{63..0}$ ← $(prod_{63})^{32}$ || $prod_{63..32}$
$GPR[rd]_{63..0}$ ← $(prod_{31})^{32}$ || $prod_{31..0}$

Exceptions:

None

Programming Notes:

See the Programming Notes for the MADD instruction

MADDU1 — Multiply-Add Unsigned word Pipeline 1 — MADDU1

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	0 00000	MADDU1 100001
6	5	5	5	5	6

C790

Format: MADDU1 rs, rt

MADDU1 rd, rs, rt

Purpose: To multiply 32-bit unsigned integers and add in Pipeline 1.

Description: (rd, HI1, LO1) ← (HI1, LO1) + rs × rt

The 32-bit value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as unsigned values, to produce a 64-bit multiply result. The 64-bit multiply result is added to the contents in special registers *HI1* (= $HI_{127..64}$) and *LO1* (= $LO_{127..64}$). The low-order 32-bit word of the result is placed into special register *LO1* and GPR *rd*, and the high-order 32-bit word of the result is placed into special register *HI1*.

No arithmetic exception occurs under any circumstances.

If GPR *rd* is omitted in assembly language, 0 is used as the default value.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain zero-extended 32-bit values (bits 63..32 equal zero), then the result of the operation will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod ← $(HI_{95..64} \| LO_{95..64}) + (0 \| GPR[rs]_{31..0}) * (0 \| GPR[rt]_{31..0})$
$LO_{127..64}$ ← $(prod_{31})^{32} \| prod_{31..0}$
$HI_{127..64}$ ← $(prod_{63})^{32} \| prod_{63..32}$
$GPR[rd]_{63..0}$ ← $(prod_{31})^{32} \| prod_{31..0}$

Exceptions:

None

Programming Notes:

See the Programming Notes for the MADD1 instruction

MFHI1 Move From HI1 Register MFHI1

31 26	25 16	15 11	10 6	5 0
MMI 011100	0 0000000000	rd	0 00000	MFHI1 010000
6	10	5	5	6

C790

Format: MFHI1 rd

Purpose: To copy the special purpose register HI1 to a GPR.

Description: rd ← HI1

The contents of special register *HI1* (= $HI_{127..64}$) are loaded into GPR *rd*.

Restrictions:

None

Operation:

$GPR[rd]_{63..0} \leftarrow HI_{127..64}$

Exceptions:

None

MFLO1 Move From LO1 Register MFLO1

31 26	25 16	15 11	10 6	5 0
MMI 011100	0 0000000000	rd	0 00000	MFLO1 010010
6	10	5	5	6

C790

Format: MFLO1 rd

Purpose: To copy the special purpose LO1 register to a GPR.

Description: rd ← LO1

The contents of special register *LO1* (= $LO_{127..64}$) are loaded into GPR *rd.*

Restrictions:

None

Operation:

$GPR[rd]_{63..0} \leftarrow LO_{127..64}$

Exceptions:

None

MFSA Move from Shift Amount Register MFSA

31 26	25 16	15 11	10 6	5 0
SPECIAL 000000	0 00 0000 0000	rd	0 00000	MFSA 101000
6	10	5	5	6

C790

Format: MFSA rd

Purpose: To copy the shift amount register SA to a GPR.

Description: rd ← SA

The contents of SA, the special register storing the funnel shift amount, is loaded into GPR *rd*. Note that the shift amount is encoded in SA in an implementation-defined manner. Therefore, it is not meaningful for software to operate on the value returned in *rd*. The sole purpose of this instruction is to permit the shift amount to be saved during a context switch. The MTSA instruction should be used to restore the state of SA.

Restrictions:

None

Operation:

$GPR[rd]_{63..0} \leftarrow SA$

Exceptions:

None

Implementation Note:

This instruction executes only in pipeline 0.

MTHI1 Move To HI1 Register MTHI1

31 26	25 21	20 6	5 0
MMI 011100	rs	0 000000000000000	MTHI1 010001
6	5	15	6

C790

Format: MTHI1 rs

Purpose: To copy a GPR to the special purpose register HI1.

Description: HI1 ← rs

The contents of GPR *rs* are loaded into special register *HI1* (= $HI_{127..64}$).

Restrictions:

None

Operation:

$HI_{127..64} \leftarrow GPR[rs]_{63..0}$

Exceptions:

None

Programming Notes:

None

MTLO1 Move To LO1 Register MTLO1

31 26	25 21	20 6	5 0
MMI 011100	rs	0 000000000000000	MTLO1 010011
6	5	15	6

C790

Format: MTLO1 rs

Purpose: To copy a GPR to the special purpose register LO1.

Description: LO1 ← rs

The contents of GPR *rs* are loaded into special register *LO1* (= $LO_{127..64}$).

Restrictions:

None

Operation:

$LO_{127..64} \leftarrow GPR[rs]_{63..0}$

Exceptions:

None

Programming Notes:

None

MTSA — Move to Shift Amount Register — MTSA

31–26	25–21	20–6	5–0
SPECIAL 000000	rs	0 000 0000 0000 0000	MTSA 101001
6	5	15	6

C790

Format: MTSA rs

Purpose: To copy a GPR to the shift amount register SA.

Description: SA ← rs

The contents of GPR *rs* are loaded into SA, the special register storing the funnel shift amount. Note that *rs* must contain a value that was originally generated by MFSA. If some other user-generated value is in *rs,* the shifting action performed by the funnel shifter is not defined; that is, MTSA cannot be used to by a program to set a new funnel shift amount. This is because the shift amount is encoded in SA in an implementation-defined manner. The sole purpose of this instruction is to permit the shift amount to be restored during a context switch.

Restrictions:

Note that the three instructions statically preceding a MTSA instruction must not read or write the SA register; that is, they cannot be either of the instructions MFSA, QFSRV, or MTSA*x*.

Use the MTSAB and MTSAH instructions to set a new funnel shift amount.

Operation:

SA ← $GPR[rs]_{63..0}$

Exceptions:

None

Implementation Note:

1. MTSA updates the SA register in the A Stage. To keep exception processing simple, this requires that the cycle prior to MTSA not read the SA register. Also, when single stepping, making sure that SA always contains the value of the SA write instruction, just single stepped, requires that the cycle after MTSA not write the SA register. Both these rules are enforced by the architectural requirement that the three instructions prior to MTSA not read SA.

2. The MTSA instruction executes only in pipeline 0.

MTSAB Move Byte Count to Shift Amount Register MTSAB

31 26	25 21	20 16	15 0
REGIMM 000001	rs	MTSAB 11000	immediate
6	5	5	16

C790

Format: MTSAB rs, immediate

Purpose: To copy a GPR to the shift amount register SA.

Description: SA ← (rs xor immediate) x 8

The least-significant four bits of GPR *rs* are XORed with the least-significant four bits of the immediate value. The resulting four bits are interpreted as a byte shift amount and stored into SA, the special register storing the funnel shift amount.

Restrictions:

The three instructions statically preceding a MTSAB instruction must not read the SA register; that is, they cannot be either of the instructions MFSA or QFSRV.

Operation:

$SA \leftarrow (GPR[rs]_{3..0} \text{ xor } immediate_{3..0}) * 8$

Exceptions:

None

Implementation Note:

1. MTSAB updates the SA register in the A Stage. To keep exception processing simple, this requires that the cycle prior to MTSAB not read the SA register. Also, when single stepping, making sure that SA always contains the value of the SA write instruction, just single stepped, requires that the cycle after the MTSAB not write the SA register. Both these rules are enforced by the architectural requirement that the three instructions prior to MTSAB not read SA.

2. The MTSAB instruction executes only in pipeline 0.

Programming Note:

MTSAB allows the user to load either a variable shift amount or a fixed shift amount, as follows:

```
mtsab   0, 5  // Set shift amount to “5 bytes”
mtsab   10, 0 // Set byte shift amount to contents of GPR10
```

MTSAH — Move Halfword Count to Shift Amount Register — MTSAH

31 26	25 21	20 16	15 0
REGIMM 000001	rs	MTSAH 11001	immediate
6	5	5	16

C790

Format: MTSAH rs, immediate

Purpose: To copy a GPR to the shift amount register SA.

Description: SA ← (rs xor immediate) x 16

The least-significant three bits of GPR *rs* are XORed with the least-significant three bits of the immediate value. The resulting three bits are interpreted as a halfword shift amount and stored into SA, the special register storing the funnel shift amount.

Restrictions:

The three instructions statically preceding a MTSAB instruction must not read the SA register; that is, they cannot be either of the instructions MFSA or QFSRV.

Operation:

SA ← (GPR[rs]$_{2..0}$ xor immediate$_{2..0}$) * 16

Exceptions:

None

Implementation Note:

1. MTSAH updates the SA register in the A Stage. To keep exception processing simple, this requires that the cycle prior to MTSAH not read the SA register. Also, when single stepping, making sure that SA always contains the value of the SA write instruction, just single stepped, requires that the cycle after MTSAH not write the SA register. Both these rules are enforced by the architectural requirement that the three instructions prior to MTSAH not read SA.

2. The MTSAH instruction executes only in pipeline 0.

Programming Note:

MTSAH allows the user to load either a variable shift amount or a fixed shift amount, as follows:

```
mtsah   0, 5  // Set shift amount to “5 halfwords”
mtsah   10, 0 // Set halfword shift amount to value of GPR10
```

MULT

Multiply Word

MULT

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	MULT 011000
6	5	5	5	5	6

C790

Format: MULT rd, rs, rt
MULT rs, rt

Purpose: To multiply 32-bit signed integers.

Description: (rd, LO, HI) ← rs × rt

The 32-bit value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as signed values, to produce a 64-bit result. The low-order 32-bits of the result is placed into special register *LO* and GPR *rd*, and the high-order 32-bit of the result is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

If GPR rd is omitted in assembly language, 0 is used as the default value.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod ← $GPR[rs]_{31..0}$ * $GPR[rt]_{31..0}$
$LO_{63..0}$ ← $(prod_{31})^{32}$ || $prod_{31..0}$
$HI_{63..0}$ ← $(prod_{63})^{32}$ || $prod_{63..32}$
$GPR[rd]_{63..0}$ ← $(prod_{31})^{32}$ || $prod_{31..0}$

Exceptions:

None

Programming Notes:

In the C790, the integer multiply operation allows other CPU instructions to execute out-of-order. An attempt to read *LO* or *HI* registers before the results are written will cause an interlock until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

MULT1 Multiply Word Pipeline 1 MULT1

31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0
MMI 011100	rs	rt	rd	0 00000	MULT1 011000
6	5	5	5	5	6

C790

Format: MULT1 rd, rs, rt
MULT1 rs, rt

Purpose: To multiply 32-bit signed integers in Pipeline 1.

Description: (rd, HI1, LO1) ← rs × rt

The 32-bit value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as signed values, to produce a 64-bit result. The low-order 32-bits of the result is placed into special register *LO1* (= $LO_{127..64}$) and GPR *rd*, and the high-order 32-bits of the result is placed into special register *HI1* (= $HI_{127..64}$).

No arithmetic exceptions occurs under any circumstances.

If GPR *rd* is omitted in assembly language, 0 is used as the default value.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod ← $GPR[rs]_{31..0}$ * $GPR[rt]_{31..0}$
$LO_{127..64}$ ← $(prod_{31})^{32}$ || $prod_{31..0}$
$HI_{127..64}$ ← $(prod_{63})^{32}$ || $prod_{63..32}$
$GPR[rd]_{63..0}$ ← $(prod_{31})^{32}$ || $prod_{31..0}$

Exceptions:

None

Programming Notes:

In the C790 the integer multiply operation allows other CPU instructions to execute out-of-order. An attempt to read *LO1* or *HI1* before the results are written will cause an interlock until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

MULTU

Multiply Unsigned Word

MULTU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	MULTU 011001
6	5	5	5	5	6

C790

Format: MULTU rd, rs, rt
MULTU rs, rt

Purpose: To multiply 32-bit unsigned integers.

Description: (rd, HI, LO) ← rs × rt

The 32-bit value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as unsigned values, to produce a 64-bit result. The low-order 32-bit of the result is placed into special register *LO* and GPR *rd*, and the high-order 32-bits of the result is placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

If GPR *rd* is omitted in assembly language, 0 is used as the default value.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain zero-extended 32-bit values (bits 63..32 equal zero), then the result of the operation will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod ← (0 ∥ $GPR[rs]_{31..0}$) * (0 ∥ $GPR[rt]_{31..0}$)
$LO_{63..0}$ ← $(prod_{31})^{32}$ ∥ $prod_{31..0}$
$HI_{63..0}$ ← $(prod_{63})^{32}$ ∥ $prod_{63..32}$
$GPR[rd]_{63..0}$ ← $(prod_{31})^{32}$ ∥ $prod_{31..0}$

Exceptions:

None

Programming Notes:

See the Programming Notes for the MULT instruction.

MULTU1 Multiply Unsigned Word Pipeline 1 MULTU1

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	0 00000	MULTU1 011001
6	5	5	5	5	6

C790

Format: MULTU1 rd, rs, rt
MULTU1 rs, rt

Purpose: To multiply 32-bit unsigned integers in Pipeline 1.

Description: (rd, HI1, LO1) ← rs × rt

The 32-bit value in GPR *rt* is multiplied by the 32-bit value in GPR *rs*, treating both operands as unsigned values, to produce a 64-bit result. The low-order 32-bit of the result is placed into special register *LO1* (= $LO_{127..64}$) and GPR *rd*, and the high-order 32-bit of the result is placed into special register *HI1* (= $HI_{127..64}$).

No arithmetic exceptions occurs under any circumstances.

If GPR rd is omitted in assembly language, 0 is used as the default value.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain zero-extended 32-bit values (bits 63..32 equal zero), then the result of the operation will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod ← (0 || $GPR[rs]_{31..0}$) * (0 || $GPR[rt]_{31..0}$)
$LO_{127..64}$ ← $(prod_{31})^{32}$ || $prod_{31..0}$
$HI_{127..64}$ ← $(prod_{63})^{32}$ || $prod_{63..32}$
$GPR[rd]_{63..0}$ ← $(prod_{31})^{32}$ || $prod_{31..0}$

Exceptions:

None

Programming Notes:

See the Programming Notes for the MULT1 instruction.

PABSH Parallel Absolute Halfword PABSH

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	0 00000	rt	rd	PABSH 00101	MMI1 101000
6	5	5	5	5	6

C790

Format: PABSH rd, rt

Purpose: To calculate the absolute value of 8 16-bit integers in parallel.

Description: rd ← |rt|

The absolute value of the eight signed halfword values in GPR *rt* are placed into the corresponding eight halfwords in GPR *rd.*

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{15..0} \leftarrow |GPR[rt]_{15..0}|$
$GPR[rd]_{31..16} \leftarrow |GPR[rt]_{31..16}|$
$GPR[rd]_{47..32} \leftarrow |GPR[rt]_{47..32}|$
$GPR[rd]_{63..48} \leftarrow |GPR[rt]_{63..48}|$
$GPR[rd]_{79..64} \leftarrow |GPR[rt]_{79..64}|$
$GPR[rd]_{95..80} \leftarrow |GPR[rt]_{95..80}|$
$GPR[rd]_{111..96} \leftarrow |GPR[rt]_{111..96}|$
$GPR[rd]_{127..112} \leftarrow |GPR[rt]_{127..112}|$

	127 112	111 96	95 80	79 64	63 48	47 32	31 16	15 0
rt	A7	A6	A5	A4	A3	A2	A1	A0

	127 112	111 96	95 80	79 64	63 48	47 32	31 16	15 0
rd	|A7|	|A6|	|A5|	|A4|	|A3|	|A2|	|A1|	|A0|

Supplementary explanation:

When the halfword value in GPR *rt* is 0x8000 (-32768), the smallest negative value, the operation will result in an overflow. However, overflow exception doesn't occur; the result is truncated to the largest positive number - 0x7FFF (+32767) .

Exceptions:

None

PABSW Parallel Absolute Word PABSW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	0 00000	rt	rd	PABSW 00001	MMI1 101000
6	5	5	5	5	6

C790

Format: PABSW rd, rt

Purpose: To calculate the absolute value of 4 32-bit integers in parallel.

Description: rd ← |rt|

The absolute value of the four signed word values in GPR *rt* are placed into the corresponding four words in GPR *rd.*

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{31..0} \leftarrow |GPR[rt]_{31..0}|$
$GPR[rd]_{63..32} \leftarrow |GPR[rt]_{63..32}|$
$GPR[rd]_{95..64} \leftarrow |GPR[rt]_{95..64}|$
$GPR[rd]_{127..96} \leftarrow |GPR[rt]_{127..96}|$

	127 96	95 64	63 32	31 0
rt	A3	A2	A1	A0

	127 96	95 64	63 32	31 0
rd	|A3|	|A2|	|A1|	|A0|

Supplementary explanation:

When the word value of the GPR *rt* is equal to 0x80000000 (-2147483648), the smallest negative number, the operation will result in an overflow. However, if an overflow exception doesn't occur; the result is truncated to the largest positive value - 0x7FFFFFFF (+2147483647).

Exceptions:

None

PADDB — Parallel Add Byte — PADDB

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PADDB 01000	MMI0 001000
6	5	5	5	5	6

C790

Format: PADDB rd, rs, rt

Purpose: To add 16 pairs of 8-bit integers in parallel.

Description: rd ← rs + rt

The sixteen byte values in GPR *rs* are added to the corresponding sixteen byte values in GPR *rt* in parallel. The results are placed into the corresponding sixteen bytes in GPR *rd*.

No overflow or underflow exceptions are generated under any circumstances. This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{7..0} \leftarrow (GPR[rs]_{7..0} + GPR[rt]_{7..0})_{7..0}$
$GPR[rd]_{15..8} \leftarrow (GPR[rs]_{15..8} + GPR[rt]_{15..8})_{7..0}$
$GPR[rd]_{23..16} \leftarrow (GPR[rs]_{23..16} + GPR[rt]_{23..16})_{7..0}$
$GPR[rd]_{31..24} \leftarrow (GPR[rs]_{31..24} + GPR[rt]_{31..24})_{7..0}$
$GPR[rd]_{39..32} \leftarrow (GPR[rs]_{39..32} + GPR[rt]_{39..32})_{7..0}$
$GPR[rd]_{47..40} \leftarrow (GPR[rs]_{47..40} + GPR[rt]_{47..40})_{7..0}$
$GPR[rd]_{55..48} \leftarrow (GPR[rs]_{55..48} + GPR[rt]_{55..48})_{7..0}$
$GPR[rd]_{63..56} \leftarrow (GPR[rs]_{63..56} + GPR[rt]_{63..56})_{7..0}$
$GPR[rd]_{71..64} \leftarrow (GPR[rs]_{71..64} + GPR[rt]_{71..64})_{7..0}$
$GPR[rd]_{79..72} \leftarrow (GPR[rs]_{79..72} + GPR[rt]_{79..72})_{7..0}$
$GPR[rd]_{87..80} \leftarrow (GPR[rs]_{87..80} + GPR[rt]_{87..80})_{7..0}$
$GPR[rd]_{95..88} \leftarrow (GPR[rs]_{95..88} + GPR[rt]_{95..88})_{7..0}$
$GPR[rd]_{103..96} \leftarrow (GPR[rs]_{103..96} + GPR[rt]_{103..96})_{7..0}$
$GPR[rd]_{111..104} \leftarrow (GPR[rs]_{111..104} + GPR[rt]_{111..104})_{7..0}$
$GPR[rd]_{119..112} \leftarrow (GPR[rs]_{119..112} + GPR[rt]_{119..112})_{7..0}$
$GPR[rd]_{127..120} \leftarrow (GPR[rs]_{127..120} + GPR[rt]_{127..120})_{7..0}$

	127–120	119–112	111–104	103–96	95–88	87–80	79–72	71–64	63–56	55–48	47–40	39–32	31–24	23–16	15–8	7–0
rs	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
rt	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
rd	A15+B15	A14+B14	A13+B13	A12+B12	A11+B11	A10+B10	A9+B9	A8+B8	A7+B7	A6+B6	A5+B5	A4+B4	A3+B3	A2+B2	A1+B1	A0+B0

Exceptions:

None

PADDH Parallel Add Halfword PADDH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PADDH 00100	MMI0 001000
6	5	5	5	5	6

C790

Format: PADDH rd, rs, rt

Purpose: To add 8 pairs of 16-bit integers in parallel.

Description: rd ← rs + rt

The eight halfword values in GPR *rs* are added to the corresponding eight halfword values in GPR *rt* in parallel. The results are placed into the corresponding eight halfwords in GPR *rd*.

No overflow or underflow exceptions are generated under any circumstances.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{15..0} \leftarrow (GPR[rs]_{15..0} + GPR[rt]_{15..0})_{15..0}$
$GPR[rd]_{31..16} \leftarrow (GPR[rs]_{31..16} + GPR[rt]_{31..16})_{15..0}$
$GPR[rd]_{47..32} \leftarrow (GPR[rs]_{47..32} + GPR[rt]_{47..32})_{15..0}$
$GPR[rd]_{63..48} \leftarrow (GPR[rs]_{63..48} + GPR[rt]_{63..48})_{15..0}$
$GPR[rd]_{79..64} \leftarrow (GPR[rs]_{79..64} + GPR[rt]_{79..64})_{15..0}$
$GPR[rd]_{95..80} \leftarrow (GPR[rs]_{95..80} + GPR[rt]_{95..80})_{15..0}$
$GPR[rd]_{111..96} \leftarrow (GPR[rs]_{111..96} + GPR[rt]_{111..96})_{15..0}$
$GPR[rd]_{127..112} \leftarrow (GPR[rs]_{127..112} + GPR[rt]_{127..112})_{15..0}$

	127–112	111–96	95–80	79–64	63–48	47–32	31–16	15–0
rs	A7	A6	A5	A4	A3	A2	A1	A0
	+	+	+	+	+	+	+	+
rt	B7	B6	B5	B4	B3	B2	B1	B0

	127–112	111–96	95–80	79–64	63–48	47–32	31–16	15–0
rd	A7+B7	A6+B6	A5+B5	A4+B4	A3+B3	A2+B2	A1+B1	A0+B0

Exceptions:

None

PADDSB   Parallel Add with Signed saturation Byte   PADDSB

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PADDSB 11000	MMI0 001000
6	5	5	5	5	6

C790

Format: PADDSB rd, rs, rt

Purpose: To add 16 pairs of 8-bit signed integers with saturation in parallel.

Description: rd ← rs + rt

The sixteen signed byte values in GPR *rs* are added to the corresponding sixteen signed byte values in GPR *rt* in parallel. The results are placed into the corresponding sixteen bytes in GPR *rd.*

No overflow or underflow exceptions are generated under any circumstances. Results beyond the range of a signed byte value are saturated according to the following:

Overflow: 0x7F

Underflow: 0x80

This instruction operates on 128-bit registers.

Operation:

```
if ((GPR[rs]7..0 + GPR[rt]7..0) > 0x7F) then
  GPR[rd]7..0  ← 0x7F
else if (0x100 <= (GPR[rs]7..0 + GPR[rt]7..0) < 0x180) then
  GPR[rd]7..0  ← 0x80
else
  GPR[rd]7..0  ← (GPR[rs]7..0 + GPR[rt]7..0)7..0
endif

if ((GPR[rs]15..8 + GPR[rt]15..8) > 0x7F) then
  GPR[rd]15..8          ← 0x7F
else if (0x100 <= (GPR[rs]15..8 + GPR[rt]15..8) < 0x180) then
  GPR[rd]15..8          ← 0x80
else
  GPR[rd]15..8          ← (GPR[rs]15..8 + GPR[rt]15..8)7..0
endif

if ((GPR[rs]23..16 + GPR[rt]23..16) > 0x7F) then
  GPR[rd]23..16         ← 0x7F
else if (0x100 <= (GPR[rs]23..16 + GPR[rt]23..16) < 0x180) then
  GPR[rd]23..16         ← 0x80
else
  GPR[rd]23..16         ← (GPR[rs]23..16 + GPR[rt]23..16)7..0
endif

if ((GPR[rs]31..24 + GPR[rt]31..24) > 0x7F) then
  GPR[rd]31..24         ← 0x7F
else if (0x100 <= (GPR[rs]31..24 + GPR[rt]31..24) < 0x180) then
```

```
  GPR[rd]31..24          ← 0x80
else
  GPR[rd]31..24          ← (GPR[rs]31..24 + GPR[rt]31..24)7..0
endif

if ((GPR[rs]39..32 + GPR[rt]39..32) > 0x7F) then
  GPR[rd]39..32          ← 0x7F
else if (0x100 <= (GPR[rs]39..32 + GPR[rt]39..32) < 0x180) then
  GPR[rd]39..32          ← 0x80
else
  GPR[rd]39..32          ← (GPR[rs]39..32 + GPR[rt]39..32)7..0
endif

if ((GPR[rs]47..40 + GPR[rt]47..40) > 0x7F) then
  GPR[rd]47..40          ← 0x7F
else if (0x100 <= (GPR[rs]47..40 + GPR[rt]47..40) < 0x180) then
  GPR[rd]47..40          ← 0x80
else
  GPR[rd]47..40          ← (GPR[rs]47..40 + GPR[rt]47..40)7..0
endif

if ((GPR[rs]55..48 + GPR[rt]55..48) > 0x7F) then
  GPR[rd]55..48          ← 0x7F
else if (0x100 <= (GPR[rs]55..48 + GPR[rt]55..48) < 0x180) then
  GPR[rd]55..48          ← 0x80
else
  GPR[rd]55..48          ← (GPR[rs]55..48 + GPR[rt]55..48)7..0
endif

if ((GPR[rs]63..56 + GPR[rt]63..56) > 0x7F) then
  GPR[rd]63..56          ← 0x7F
else if (0x100 <= (GPR[rs]63..56 + GPR[rt]63..56) < 0x180) then
  GPR[rd]63..56          ← 0x80
else
  GPR[rd]63..56          ← (GPR[rs]63..56 + GPR[rt]63..56)7..0
endif

if ((GPR[rs]71..64 + GPR[rt]71..64) > 0x7F) then
  GPR[rd]71..64          ← 0x7F
else if (0x100 <= (GPR[rs]71..64 + GPR[rt]71..64) < 0x180) then
  GPR[rd]71..64          ← 0x80
else
  GPR[rd]71..64          ← (GPR[rs]71..64 + GPR[rt]71..64)7..0
endif

if ((GPR[rs]79..72 + GPR[rt]79..72) > 0x7F) then
  GPR[rd]79..72          ← 0x7F
else if (0x100 <= (GPR[rs]79..72 + GPR[rt]79..72) < 0x180) then
  GPR[rd]79..72          ← 0x80
else
  GPR[rd]79..72          ← (GPR[rs]79..72 + GPR[rt]79..72)7..0
endif

if ((GPR[rs]87..80 + GPR[rt]87..80) > 0x7F) then
  GPR[rd]87..80          ← 0x7F
```

```
else if (0x100 <= (GPR[rs]87..80 + GPR[rt]87..80) < 0x180) then
  GPR[rd]87..80          ← 0x80
else
  GPR[rd]87..80          ← (GPR[rs]87..80 + GPR[rt]87..80)7..0
endif

if ((GPR[rs]95..88 + GPR[rt]95..88) > 0x7F) then
  GPR[rd]95..88          ← 0x7F
else if (0x100 <= (GPR[rs]95..88 + GPR[rt]95..88) < 0x180) then
  GPR[rd]95..88          ← 0x80
else
  GPR[rd]95..88          ← (GPR[rs]95..88 + GPR[rt]95..88)7..0
endif

if ((GPR[rs]103..96 + GPR[rt]103..96) > 0x7F) then
  GPR[rd]103..96         ← 0x7F
else if (0x100 <= (GPR[rs]103..96 + GPR[rt]103..96) < 0x180) then
  GPR[rd]103..96         ← 0x80
else
  GPR[rd]103..96         ← (GPR[rs]103..96 + GPR[rt]103..96)7..0
endif

if ((GPR[rs]111..104 + GPR[rt]111..104) > 0x7F) then
  GPR[rd]111..104        ← 0x7F
else if (0x100 <= (GPR[rs]111..104 + GPR[rt]111..104) < 0x180) then
  GPR[rd]111..104        ← 0x80
else
  GPR[rd]111..104        ← (GPR[rs]111..104 + GPR[rt]111..104)7..0
endif

if ((GPR[rs]119..112 + GPR[rt]119..112) > 0x7F) then
  GPR[rd]119..112        ← 0x7F
else if (0x100 <= (GPR[rs]119..112 + GPR[rt]119..112) < 0x180) then
  GPR[rd]119..112        ← 0x80
else
  GPR[rd]119..112        ← (GPR[rs]119..112 + GPR[rt]119..112)7..0
endif

if ((GPR[rs]127..120 + GPR[rt]127..120) > 0x7F) then
  GPR[rd]127..120        ← 0x7F
else if (0x100 <= (GPR[rs]127..120 + GPR[rt]127..120) < 0x180) then
  GPR[rd]127..120        ← 0x80
else
  GPR[rd]127..120        ← (GPR[rs]127..120 + GPR[rt]127..120)7..0
endif
```

	127 120	119 112	111 104	103 96	95 88	87 80	79 72	71 64	63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
rs	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
rt	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
rd*	A15 + B15	A14 + B14	A13 + B13	A12 + B12	A11 + B11	A10 + B10	A9 + B9	A8 + B8	A7 + B7	A6 + B6	A5 + B5	A4 + B4	A3 + B3	A2 + B2	A1 + B1	A0 + B0

* Saturate to signed byte

Exceptions:

None

PADDSH Parallel Add with Signed saturation Halfword PADDSH

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PADDSH 10100	MMI0 001000
6	5	5	5	5	6

C790

Format: PADDSH rd, rs, rt

Purpose: To add 8 pairs of 16-bit signed integers with saturation in parallel.

Description: rd ← rs + rt

The eight signed halfword values in GPR *rs* are added to the corresponding eight signed halfword values in GPR *rt* in parallel. The results are placed into the corresponding eight halfwords in GPR *rd.*

No overflow or underflow exceptions are generated under any circumstances. Results beyond the range of a signed halfword value are saturated according to the following:

Overflow: 0x7FFF

Underflow: 0x8000

This instruction operates on 128-bit registers.

Operation:

```
if ((GPR[rs]15..0 + GPR[rt]15..0) > 0x7FFF) then
  GPR[rd]15..0        ← 0x7FFF
else if (0x10000 <= (GPR[rs]15..0 + GPR[rt]15..0) < 0x18000) then
  GPR[rd]15..0        ← 0x8000
else
  GPR[rd]15..0        ← (GPR[rs]15..0 + GPR[rt]15..0)15..0
endif

if ((GPR[rs]31..16 + GPR[rt]31..16) > 0x7FFF) then
  GPR[rd]31..16       ← 0x7FFF
else if (0x10000 <= (GPR[rs]31..16 + GPR[rt]31..16) < 0x18000) then
  GPR[rd]31..16       ← 0x8000
else
  GPR[rd]31..16       ← (GPR[rs]31..16 + GPR[rt]31..16)15..0
endif

if ((GPR[rs]47..32 + GPR[rt]47..32) > 0x7FFF) then
  GPR[rd]47..32       ← 0x7FFF
else if (0x10000 <= (GPR[rs]47..32 + GPR[rt]47..32) < 0x18000) then
  GPR[rd]47..32       ← 0x8000
else
  GPR[rd]47..32       ← (GPR[rs]47..32 + GPR[rt]47..32)15..0
endif
```

```
if ((GPR[rs]63..48 + GPR[rt]63..48) > 0x7FFF) then
  GPR[rd]63..48          ← 0x7FFF
else if (0x10000 <= (GPR[rs]63..48 + GPR[rt]63..48) < 0x18000) then
  GPR[rd]63..48          ← 0x8000
else
  GPR[rd]63..48          ← (GPR[rs]63..48 + GPR[rt]63..48)15..0
endif

if ((GPR[rs]79..64 + GPR[rt]79..64) > 0x7FFF) then
  GPR[rd]79..64          ← 0x7FFF
else if (0x10000 <= (GPR[rs]79..64 + GPR[rt]79..64) < 0x18000) then
  GPR[rd]79..64          ← 0x8000
else
  GPR[rd]79..64          ← (GPR[rs]79..64 + GPR[rt]79..64)15..0
endif

if ((GPR[rs]95..80 + GPR[rt]95..80) > 0x7FFF) then
  GPR[rd]95..80          ← 0x7FFF
else if (0x10000 <= (GPR[rs]95..80 + GPR[rt]95..80) < 0x18000) then
  GPR[rd]95..80          ← 0x8000
else
  GPR[rd]95..80          ← (GPR[rs]95..80 + GPR[rt]95..80)15..0
endif

if ((GPR[rs]111..96 + GPR[rt]111..96) > 0x7FFF) then
  GPR[rd]111..96         ← 0x7FFF
else if (0x10000 <= (GPR[rs]111..96 + GPR[rt]111..96) < 0x18000) then
  GPR[rd]111..96         ← 0x8000
else
  GPR[rd]111..96         ← (GPR[rs]111..96 + GPR[rt]111..96)15..0
endif

if ((GPR[rs]127..112 + GPR[rt]127..112) > 0x7FFF) then
  GPR[rd]127..112        ← 0x7FFF
else if (0x10000 <= (GPR[rs]127..112 + GPR[rt]127..112) < 0x18000) then
  GPR[rd]127..112        ← 0x8000
else
  GPR[rd]127..112        ← (GPR[rs]127..112 + GPR[rt]127..112)15..0
endif
```

	127 112	111 96	95 80	79 64	63 48	47 32	31 16	15 0
rs	A7	A6	A5	A4	A3	A2	A1	A0
	+	+	+	+	+	+	+	+
rt	B7	B6	B5	B4	B3	B2	B1	B0

	127 112	111 96	95 80	79 64	63 48	47 32	31 16	15 0
rd*	A7+B7	A6+B6	A5+B5	A4+B4	A3+B3	A2+B2	A1+B1	A0+B0

* Saturate to signed halfword

Exceptions:

None

PADDSW

Parallel Add with Signed saturation Word

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PADDSW 10000	MMI0 001000
6	5	5	5	5	6

C790

Format: PADDSW rd, rs, rt

Purpose: To add 4 pairs of 32-bit signed integers with saturation in parallel.

Description: rd ← rs + rt

The four signed word values in GPR *rs* are added to the corresponding four signed word values in GPR *rt* in parallel. The results are placed into to the corresponding four words in GPR *rd*.

No overflow or underflow exceptions are generated under any circumstances. Results beyond the range of a signed word value are saturated according to the following:

Overflow:	0x7FFFFFFF
Underflow:	0x80000000

This instruction operates on 128-bit registers.

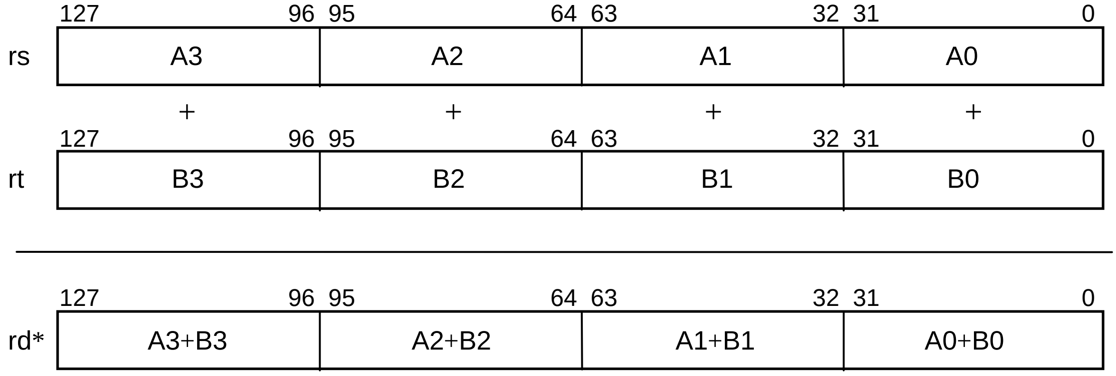
Operation:

```
if ((GPR[rs]31..0 + GPR[rt]31..0) > 0x7FFFFFFF) then
  GPR[rd]31..0          ← 0x7FFFFFFF
else if (0x100000000 <= (GPR[rs]31..0 + GPR[rt]31..0) < 0x180000000) then
  GPR[rd]31..0          ← 0x80000000
else
  GPR[rd]31..0          ← (GPR[rs]31..0 + GPR[rt]31..0)31..0
endif

if ((GPR[rs]63..32 + GPR[rt]63..32) > 0x7FFFFFFF) then
  GPR[rd]63..32         ← 0x7FFFFFFF
else if (0x100000000 <= (GPR[rs]63..32 + GPR[rt]63..32) < 0x180000000) then
  GPR[rd]63..32         ← 0x80000000
else
  GPR[rd]63..32         ← (GPR[rs]63..32 + GPR[rt]63..32)31..0
endif

if ((GPR[rs]95..64 + GPR[rt]95..64) > 0x7FFFFFFF) then
  GPR[rd]95..64         ← 0x7FFFFFFF
else if (0x100000000 <= (GPR[rs]95..64 + GPR[rt]95..64) < 0x180000000) then
  GPR[rd]95..64         ← 0x80000000
else
  GPR[rd]95..64         ← (GPR[rs]95..64 + GPR[rt]95..64)31..0
endif
```

```
if ((GPR[rs]127..96 + GPR[rt]127..96) > 0x7FFFFFFF) then
  GPR[rd]127..96          ← 0x7FFFFFFF
else if (0x100000000 <= (GPR[rs]127..96 + GPR[rt]127..96) < 0x180000000) then
  GPR[rd]127..96          ← 0x80000000
else
  GPR[rd]127..96          ← (GPR[rs]127..96 + GPR[rt]127..96)31..0
endif
```



* Saturate to signed word

Exceptions:

None

PADDUB Parallel Add with Unsigned saturation Byte PADDUB

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PADDUB 11000	MMI1 101000
6	5	5	5	5	6

C790

Format: PADDUB rd, rs, rt

Purpose: To add 16 pairs of 8-bit unsigned integers with saturation in parallel.

Description: rd ← rs + rt

The sixteen unsigned byte values in GPR *rs* are added to the corresponding sixteen unsigned byte values in GPR *rt* in parallel. The results are placed into the corresponding sixteen bytes in GPR *rd*.

No overflow exceptions are generated under any circumstances. Results beyond the range of an unsigned byte value are saturated according to the following:

Overflow: 0xFF

This instruction operates on 128-bit registers.

Operation:

```
if ((GPR[rs]7..0 + GPR[rt]7..0) > 0xFF) then
  GPR[rd]7..0  ← 0xFF
else
  GPR[rd]7..0  ← (GPR[rs]7..0 + GPR[rt]7..0)7..0
endif

if ((GPR[rs]15..8 + GPR[rt]15..8) > 0xFF) then
  GPR[rd]15..8        ← 0xFF
else
  GPR[rd]15..8        ← (GPR[rs]15..8 + GPR[rt]15..8)7..0
endif

if ((GPR[rs]23..16 + GPR[rt]23..16) > 0xFF) then
  GPR[rd]23..16       ← 0xFF
else
  GPR[rd]23..16       ← (GPR[rs]23..16 + GPR[rt]23..16)7..0
endif

if ((GPR[rs]31..24 + GPR[rt]31..24) > 0xFF) then
  GPR[rd]31..24       ← 0xFF
else
  GPR[rd]31..24       ← (GPR[rs]31..24 + GPR[rt]31..24)7..0
endif

if ((GPR[rs]39..32 + GPR[rt]39..32) > 0xFF) then
  GPR[rd]39..32       ← 0xFF
else
  GPR[rd]39..32       ← (GPR[rs]39..32 + GPR[rt]39..32)7..0
endif
```

```
if ((GPR[rs]47..40 + GPR[rt]47..40) > 0xFF) then
  GPR[rd]47..40          ← 0xFF
else
  GPR[rd]47..40          ← (GPR[rs]47..40 + GPR[rt]47..40)7..0
endif

if ((GPR[rs]55..48 + GPR[rt]55..48) > 0xFF) then
  GPR[rd]55..48          ← 0xFF
else
  GPR[rd]55..48          ← (GPR[rs]55..48 + GPR[rt]55..48)7..0
endif

if ((GPR[rs]63..56 + GPR[rt]63..56) > 0xFF) then
  GPR[rd]63..56          ← 0xFF
else
  GPR[rd]63..56          ← (GPR[rs]63..56 + GPR[rt]63..56)7..0
endif

if ((GPR[rs]71..64 + GPR[rt]71..64) > 0xFF) then
  GPR[rd]71..64          ← 0xFF
else
  GPR[rd]71..64          ← (GPR[rs]71..64 + GPR[rt]71..64)7..0
endif

if ((GPR[rs]79..72 + GPR[rt]79..72) > 0xFF) then
  GPR[rd]79..72          ← 0xFF
else
  GPR[rd]79..72          ← (GPR[rs]79..72 + GPR[rt]79..72)7..0
endif

if ((GPR[rs]87..80 + GPR[rt]87..80) > 0xFF) then
  GPR[rd]87..80          ← 0xFF
else
  GPR[rd]87..80          ← (GPR[rs]87..80 + GPR[rt]87..80)7..0
endif

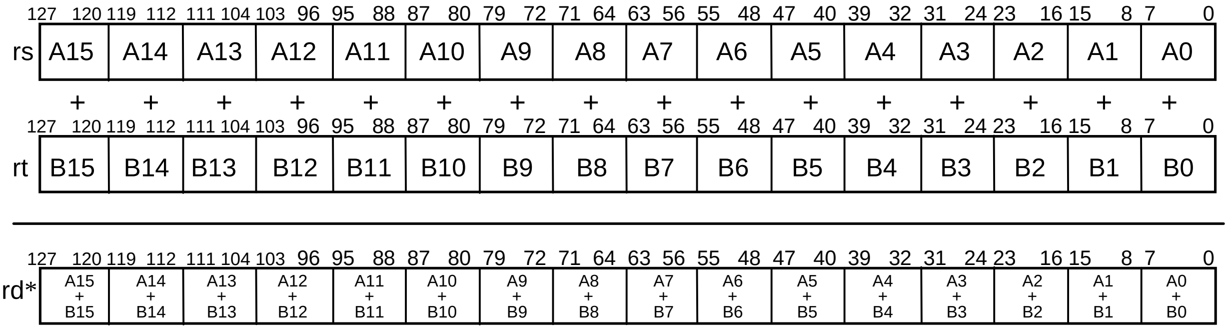
if ((GPR[rs]95..88 + GPR[rt]95..88) > 0xFF) then
  GPR[rd]95..88          ← 0xFF
else
  GPR[rd]95..88          ← (GPR[rs]95..88 + GPR[rt]95..88)7..0
endif

if ((GPR[rs]103..96 + GPR[rt]103..96) > 0xFF) then
  GPR[rd]103..96         ← 0xFF
else
  GPR[rd]103..96         ← (GPR[rs]103..96 + GPR[rt]103..96)7..0
endif
if ((GPR[rs]111..104 + GPR[rt]111..104) > 0xFF) then
  GPR[rd]111..104        ← 0xFF
else
  GPR[rd]111..104        ← (GPR[rs]111..104 + GPR[rt]111..104)7..0
endif

if ((GPR[rs]119..112 + GPR[rt]119..112) > 0xFF) then
```

$GPR[rd]_{119..112} \leftarrow 0xFF$
else
$GPR[rd]_{119..112} \leftarrow (GPR[rs]_{119..112} + GPR[rt]_{119..112})_{7..0}$
endif

if $((GPR[rs]_{127..120} + GPR[rt]_{127..120}) > 0xFF)$ then
$GPR[rd]_{127..120} \leftarrow 0xFF$
else
$GPR[rd]_{127..120} \leftarrow (GPR[rs]_{127..120} + GPR[rt]_{127..120})_{7..0}$
endif



* Saturate to unsigned byte

Exceptions:

None

PADDUH Parallel Add with Unsigned saturation Halfword PADDUH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PADDUH 10100	MMI1 101000
6	5	5	5	5	6

C790

Format: PADDUH rd, rs, rt

Purpose: To add 8 pairs of 16-bit unsigned integers with saturation in parallel.

Description: rd ← rs + rt

The eight unsigned halfword values in GPR *rs* are added to the corresponding eight unsigned halfword values in GPR *rt* in parallel. The results are placed into the corresponding eight halfwords in GPR *rd.*

No overflow exceptions are generated under any circumstances. Results beyond the range of an unsigned halfword value are saturated according to the following:

Overflow: 0xFFFF

This instruction operates on 128-bit registers.

Operation:

```
if ((GPR[rs]15..0 + GPR[rt]15..0) > 0xFFFF) then
  GPR[rd]15..0        ← 0xFFFF
else
  GPR[rd]15..0        ← (GPR[rs]15..0 + GPR[rt]15..0)15..0
endif

if ((GPR[rs]31..16 + GPR[rt]31..16) > 0xFFFF) then
  GPR[rd]31..16       ← 0xFFFF
else
  GPR[rd]31..16       ← (GPR[rs]31..16 + GPR[rt]31..16)15..0
endif

if ((GPR[rs]47..32 + GPR[rt]47..32) > 0xFFFF) then
  GPR[rd]47..32       ← 0xFFFF
else
  GPR[rd]47..32       ← (GPR[rs]47..32 + GPR[rt]47..32)15..0
endif

if ((GPR[rs]63..48 + GPR[rt]63..48) > 0xFFFF) then
  GPR[rd]63..48       ← 0xFFFF
else
  GPR[rd]63..48       ← (GPR[rs]63..48 + GPR[rt]63..48)15..0
endif
```

if ((GPR[rs]$_{79..64}$ + GPR[rt]$_{79..64}$) > 0xFFFF) then
  GPR[rd]$_{79..64}$ ← 0xFFFF
else
  GPR[rd]$_{79..64}$ ← (GPR[rs]$_{79..64}$ + GPR[rt]$_{79..64}$)$_{15..0}$
endif

if ((GPR[rs]$_{95..80}$ + GPR[rt]$_{95..80}$) > 0xFFFF) then
  GPR[rd]$_{95..80}$ ← 0xFFFF
else
  GPR[rd]$_{95..80}$ ← (GPR[rs]$_{95..80}$ + GPR[rt]$_{95..80}$)$_{15..0}$
endif

if ((GPR[rs]$_{111..96}$ + GPR[rt]$_{111..96}$) > 0xFFFF) then
  GPR[rd]$_{111..96}$ ← 0xFFFF
else
  GPR[rd]$_{111..96}$ ← (GPR[rs]$_{111..96}$ + GPR[rt]$_{111..96}$)$_{15..0}$
endif

if ((GPR[rs]$_{127..112}$ + GPR[rt]$_{127..112}$) > 0xFFFF) then
  GPR[rd]$_{127..112}$ ← 0xFFFF
else
  GPR[rd]$_{127..112}$ ← (GPR[rs]$_{127..112}$ + GPR[rt]$_{127..112}$)$_{15..0}$
endif

	127 112	111 96	95 80	79 64	63 48	47 32	31 16	15 0
rs	A7	A6	A5	A4	A3	A2	A1	A0
	+	+	+	+	+	+	+	+
rt	B7	B6	B5	B4	B3	B2	B1	B0
rd*	A7+B7	A6+B6	A5+B5	A4+B4	A3+B3	A2+B2	A1+B1	A0+B0

* Saturate to unsigned halfword

Exceptions:

None

PADDUW Parallel Add with Unsigned saturation Word PADDUW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PADDUW 10000	MMI1 101000
6	5	5	5	5	6

C790

Format: PADDUW rd, rs, rt

Purpose: To add 4 pairs of 32-bit unsigned integers with saturation in parallel.

Description: rd ← rs + rt

The four unsigned word values in GPR *rs* are added to the corresponding four unsigned word values in GPR *rt* in parallel. The results are placed into the corresponding four words in GPR *rd*.

No overflow exceptions are generated under any circumstances. Results beyond the range of an unsigned word value are saturated according to the following:

Overflow: 0xFFFFFFFF

This instruction operates on 128-bit registers.

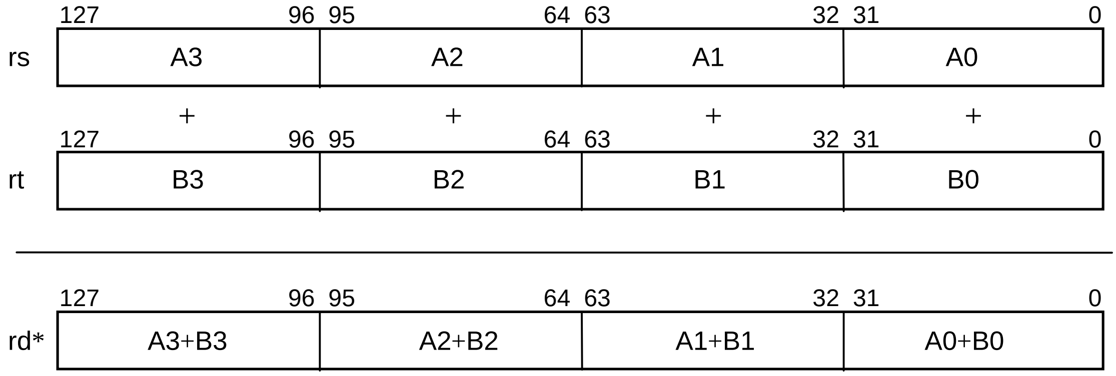
Operation:

```
if ((GPR[rs]31..0 + GPR[rt]31..0) > 0xFFFFFFFF) then
  GPR[rd]31..0        ← 0xFFFFFFFF
else
  GPR[rd]31..0        ← (GPR[rs]31..0 + GPR[rt]31..0)31..0
endif

if ((GPR[rs]63..32 + GPR[rt]63..32) > 0xFFFFFFFF) then
  GPR[rd]63..32       ← 0xFFFFFFFF
else
  GPR[rd]63..32       ← (GPR[rs]63..32 + GPR[rt]63..32)31..0
endif

if ((GPR[rs]95..64 + GPR[rt]95..64) > 0xFFFFFFFF) then
  GPR[rd]95..64       ← 0xFFFFFFFF
else
  GPR[rd]95..64       ← (GPR[rs]95..64 + GPR[rt]95..64)31..0
endif

if ((GPR[rs]127..96 + GPR[rt]127..96) > 0xFFFFFFFF) then
  GPR[rd]127..96      ← 0xFFFFFFFF
else
  GPR[rd]127..96      ← (GPR[rs]127..96 + GPR[rt]127..96)31..0
endif
```



* Saturate to unsigned word

Exceptions:

None

PADDW Parallel Add Word PADDW

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PADDW 00000	MMI0 001000
6	5	5	5	5	6

C790

Format: PADDW rd, rs, rt

Purpose: To add 4 pairs of 32-bit integers in parallel.

Description: rd ← rs + rt

The four word values in GPR *rs* are added to the corresponding four word values in GPR *rt* in parallel. The results are placed into the corresponding four words in GPR *rd*.

No overflow or underflow exceptions are generated under any circumstances.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{31..0} \leftarrow (GPR[rs]_{31..0} + GPR[rt]_{31..0})_{31..0}$
$GPR[rd]_{63..32} \leftarrow (GPR[rs]_{63..32} + GPR[rt]_{63..32})_{31..0}$
$GPR[rd]_{95..64} \leftarrow (GPR[rs]_{95..64} + GPR[rt]_{95..64})_{31..0}$
$GPR[rd]_{127..96} \leftarrow (GPR[rs]_{127..96} + GPR[rt]_{127..96})_{31..0}$

	127–96	95–64	63–32	31–0
rs	A3	A2	A1	A0
	+	+	+	+
rt	B3	B2	B1	B0
rd	A3+B3	A2+B2	A1+B1	A0+B0

Exceptions:

None

PADSBH Parallel Add/Subtract Halfword PADSBH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PADSBH 00100	MMI1 101000
6	5	5	5	5	6

C790

Format: PADSBH rd, rs, rt

Purpose: To add/subtract 8 pairs of 16-bit integers in parallel.

Description: rd ← rs +/– rt

The high-order four halfword values in GPR *rs* are added to the corresponding four halfword values in GPR *rt* and the low-order four halfword values in GPR *rt* are subtracted from the corresponding four halfword values in GPR *rs* in parallel. The results are placed into the corresponding eight halfword values in GPR *rd*.

No overflow or underflow exceptions are generated under any circumstances.

This instruction operates on 128-bit registers.

Operation

$GPR[rd]_{15..0} \leftarrow (GPR[rs]_{15..0} - GPR[rt]_{15..0})_{15..0}$
$GPR[rd]_{31..16} \leftarrow (GPR[rs]_{31..16} - GPR[rt]_{31..16})_{15..0}$
$GPR[rd]_{47..32} \leftarrow (GPR[rs]_{47..32} - GPR[rt]_{47..32})_{15..0}$
$GPR[rd]_{63..48} \leftarrow (GPR[rs]_{63..48} - GPR[rt]_{63..48})_{15..0}$
$GPR[rd]_{79..64} \leftarrow (GPR[rs]_{79..64} + GPR[rt]_{79..64})_{15..0}$
$GPR[rd]_{95..80} \leftarrow (GPR[rs]_{95..80} + GPR[rt]_{95..80})_{15..0}$
$GPR[rd]_{111..96} \leftarrow (GPR[rs]_{111..96} + GPR[rt]_{111..96})_{15..0}$
$GPR[rd]_{127..112} \leftarrow (GPR[rs]_{127..112} + GPR[rt]_{127..112})_{15..0}$

	127–112	111–96	95–80	79–64	63–48	47–32	31–16	15–0
rs	A7	A6	A5	A4	A3	A2	A1	A0
	+	+	+	+	–	–	–	–
rt	B7	B6	B5	B4	B3	B2	B1	B0
rd	A7+B7	A6+B6	A5+B5	A4+B4	A3–B3	A2–B2	A1–B1	A0–B0

Exceptions:

None

PAND Parallel And PAND

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PAND 10010	MMI2 001001
6	5	5	5	5	6

C790

Format: PAND rd, rs, rt

Purpose: To perform a bitwise logical AND.

Description: rd ← rs AND rt

The contents of GPR *rs* are combined with the contents of GPR rt in a bitwise logical AND operation. The result is placed into GPR *rd*.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{127..0} \leftarrow GPR[rs]_{127..0}$ and $GPR[rt]_{127..0}$

	127–64	63–0
rs	A1	A0
	AND	AND
rt	B1	B0
rd	A1 AND B1	A0 AND B0

Exceptions:

None

PCEQB Parallel Compare for Equal Byte PCEQB

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PCEQB 01010	MMI1 101000
6	5	5	5	5	6

C790

Format: PCEQB rd, rs, rt

Purpose: To record the result of 16 equality comparisons in parallel.

Description: rd ← (rs = rt)

The sixteen signed byte values in GPR *rs* are compared to the corresponding sixteen signed byte values in GPR *rt*, in parallel. The results of the comparison are placed into GPR *rd* as follows:

If the signed byte value in GPR *rs* is equal to the corresponding signed byte value in GPR *rt*, then the corresponding byte in GPR *rd* is set to 0xFF otherwise it is set to 0x00.

This instruction operates on 128-bit registers.

Operation:

if (GPR[rs]$_{7..0}$ = GPR[rt]$_{7..0}$) then
  GPR[rd]$_{7..0}$ ← 1^8
else
  GPR[rd]$_{7..0}$ ← 0^8
endif

if (GPR[rs]$_{15..8}$ = GPR[rt]$_{15..8}$) then
  GPR[rd]$_{15..8}$ ← 1^8
else
  GPR[rd]$_{15..8}$ ← 0^8
endif

if (GPR[rs]$_{23..16}$ = GPR[rt]$_{23..16}$) then
  GPR[rd]$_{23..16}$ ← 1^8
else
  GPR[rd]$_{23..16}$ ← 0^8
endif

if (GPR[rs]$_{31..24}$ = GPR[rt]$_{31..24}$) then
  GPR[rd]$_{31..24}$ ← 1^8
else
  GPR[rd]$_{31..24}$ ← 0^8
endif

```
if (GPR[rs]39..32 = GPR[rt]39..32) then
  GPR[rd]39..32 ← 1^8
else
  GPR[rd]39..32 ← 0^8
endif

if (GPR[rs]47..40 = GPR[rt]47..40) then
  GPR[rd]47..40 ← 1^8
else
  GPR[rd]47..40 ← 0^8
endif

if (GPR[rs]55..48 = GPR[rt]55..48) then
  GPR[rd]55..48 ← 1^8
else
  GPR[rd]55..48 ← 0^8
endif

if (GPR[rs]63..56 = GPR[rt]63..56) then
  GPR[rd]63..56 ← 1^8
else
  GPR[rd]63..56 ← 0^8
endif

if (GPR[rs]71..64 = GPR[rt]71..64) then
  GPR[rd]71..64 ← 1^8
else
  GPR[rd]71..64 ← 0^8
endif

if (GPR[rs]79..72 = GPR[rt]79..72) then
  GPR[rd]79..72 ← 1^8
else
  GPR[rd]79..72 ← 0^8
endif

if (GPR[rs]87..80 = GPR[rt]87..80) then
  GPR[rd]87..80 ← 1^8
else
  GPR[rd]87..80 ← 0^8
endif

if (GPR[rs]95..88 = GPR[rt]95..88) then
  GPR[rd]95..88 ← 1^8
else
  GPR[rd]95..88 ← 0^8
endif

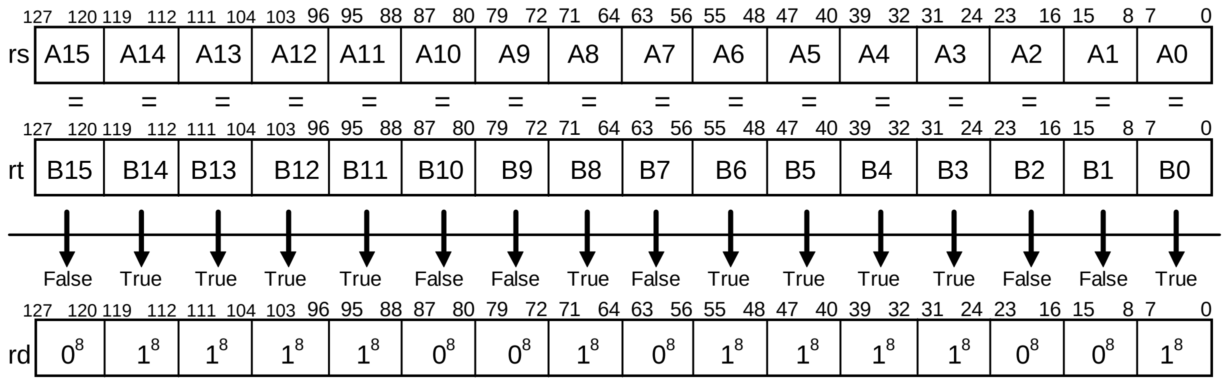
if (GPR[rs]103..96 = GPR[rt]103..96) then
  GPR[rd]103..96 ← 1^8
else
  GPR[rd]103..96 ← 0^8
endif

if (GPR[rs]111..104 = GPR[rt]111..104) then
```

GPR[rd]$_{111..104}$ ← 1^8
else
GPR[rd]$_{111..104}$ ← 0^8
endif

if (GPR[rs]$_{119..112}$ = GPR[rt]$_{119..112}$) then
GPR[rd]$_{119..112}$ ← 1^8
else
GPR[rd]$_{119..112}$ ← 0^8
endif

if (GPR[rs]$_{127..120}$ = GPR[rt]$_{127..120}$) then
GPR[rd]$_{127..120}$ ← 1^8
else
GPR[rd]$_{127..120}$ ← 0^8
endif



Exceptions:

None

PCEQH   Parallel Compare for Equal Halfword   PCEQH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PCEQH 00110	MMI1 101000
6	5	5	5	5	6

C790

Format: PCEQH rd, rs, rt

Purpose: To record the results of 8 equality comparisons in parallel.

Description: rd ← (rs = rt)

The eight signed halfword values in GPR *rs* are compared to the corresponding eight signed halfword values in GPR *rt*, in parallel. The results of the comparison are placed into GPR *rd* as follows:

If the signed halfword value in GPR *rs* is equal to the corresponding signed halfword value in GPR *rt*, then the corresponding halfword in GPR *rd* is set to 0xFFFF otherwise it is set to 0x0000.

This instruction operates on 128-bit registers.

Operation:

if (GPR[rs]$_{15..0}$ = GPR[rt]$_{15..0}$) then
  GPR[rd]$_{15..0}$ ← 1^{16}
else
  GPR[rd]$_{15..0}$ ← 0^{16}
endif

if (GPR[rs]$_{31..16}$ = GPR[rt]$_{31..16}$) then
  GPR[rd]$_{31..16}$ ← 1^{16}
else
  GPR[rd]$_{31..16}$ ← 0^{16}
endif

if (GPR[rs]$_{47..32}$ = GPR[rt]$_{47..32}$) then
  GPR[rd]$_{47..32}$ ← 1^{16}
else
  GPR[rd]$_{47..32}$ ← 0^{16}
endif

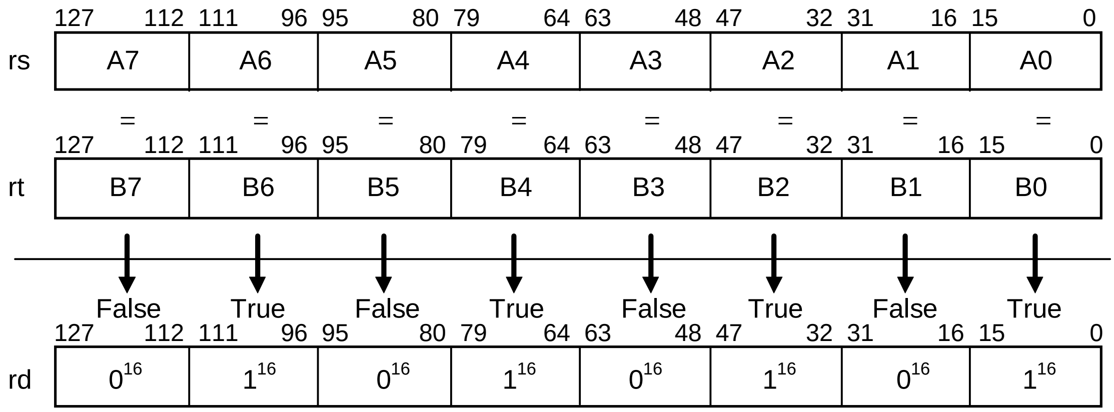
if (GPR[rs]$_{63..48}$ = GPR[rt]$_{63..48}$) then
  GPR[rd]$_{63..48}$ ← 1^{16}
else
  GPR[rd]$_{63..48}$ ← 0^{16}
endif

if $(GPR[rs]_{79..64} = GPR[rt]_{79..64})$ then
  $GPR[rd]_{79..64} \leftarrow 1^{16}$
else
  $GPR[rd]_{79..64} \leftarrow 0^{16}$
endif

if $(GPR[rs]_{95..80} = GPR[rt]_{95..80})$ then
  $GPR[rd]_{95..80} \leftarrow 1^{16}$
else
  $GPR[rd]_{95..80} \leftarrow 0^{16}$
endif

if $(GPR[rs]_{111..96} = GPR[rt]_{111..96})$ then
  $GPR[rd]_{111..96} \leftarrow 1^{16}$
else
  $GPR[rd]_{111..96} \leftarrow 0^{16}$
endif

if $(GPR[rs]_{127..112} = GPR[rt]_{127..112})$ then
  $GPR[rd]_{127..112} \leftarrow 1^{16}$
else
  $GPR[rd]_{127..112} \leftarrow 0^{16}$
endif



Exceptions:

None

PCEQW Parallel Compare for Equal Word PCEQW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PCEQW 00010	MMI1 101000
6	5	5	5	5	6

C790

Format: PCEQW rd, rs, rt

Purpose: To record the result of 4 equality comparisons in parallel.

Description: rd ← (rs = rt)

The four signed word values in GPR *rs* are compared to the corresponding four signed word values in GPR *rt*, in parallel. The results of the comparison are placed into GPR *rd* as follows:

If the signed word value in GPR *rs* is equal to the corresponding signed word value in GPR *rt*, then the corresponding word in GPR *rd* is set to 0xFFFFFFFF otherwise it is set to 0x00000000.

This instruction operates on 128-bit registers.

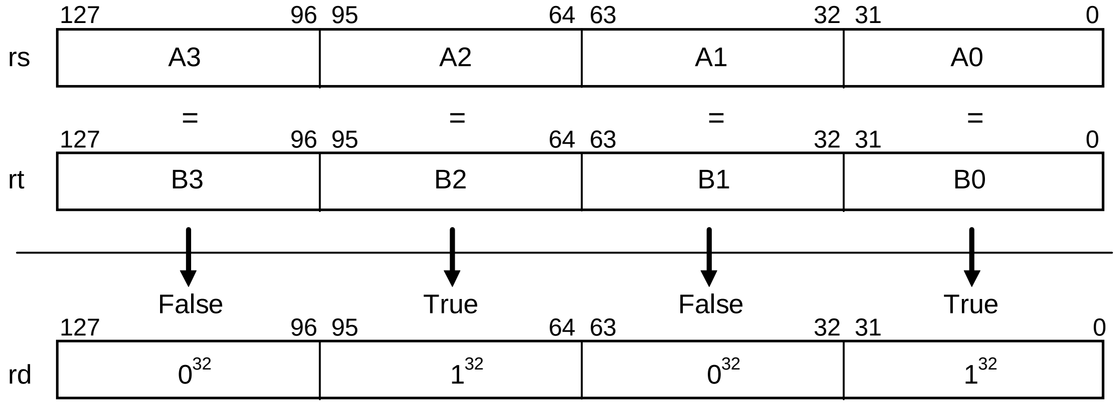
Operation:

if ($GPR[rs]_{31..0}$ = $GPR[rt]_{31..0}$) then
  $GPR[rd]_{31..0} \leftarrow 1^{32}$
else
  $GPR[rd]_{31..0} \leftarrow 0^{32}$
endif

if ($GPR[rs]_{63..32}$ = $GPR[rt]_{63..32}$) then
  $GPR[rd]_{63..32} \leftarrow 1^{32}$
else
  $GPR[rd]_{63..32} \leftarrow 0^{32}$
endif

if ($GPR[rs]_{95..64}$ = $GPR[rt]_{95..64}$) then
  $GPR[rd]_{95..64} \leftarrow 1^{32}$
else
  $GPR[rd]_{95..64} \leftarrow 0^{32}$
endif

if ($GPR[rs]_{127..96}$ = $GPR[rt]_{127..96}$) then
  $GPR[rd]_{127..96} \leftarrow 1^{32}$
else
  $GPR[rd]_{127..96} \leftarrow 0^{32}$
endif



Exceptions:

None

PCGTB Parallel Compare for Greater Than Byte PCGTB

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PCGTB 01010	MMI0 001000
6	5	5	5	5	6

C790

Format: PCGTB rd, rs, rt

Purpose: To record the result of 16 greater-than comparisons in parallel.

Description: rd ← (rs > rt)

The sixteen signed byte values in GPR *rs* are compared to the corresponding sixteen signed byte values in GPR *rt* in parallel. The results of the comparison are placed into GPR *rd* as follows:

If the signed byte value in GPR *rs* is greater than the corresponding signed byte value in GPR *rt*, then the corresponding byte in GPR *rd* is set to 0xFF otherwise it is set to 0x00.

This instruction operates on 128-bit registers.

Operation:

```
if (GPR[rs]7..0 > GPR[rt]7..0) then
  GPR[rd]7..0 ← 1^8
else
  GPR[rd]7..0 ← 0^8
endif

if (GPR[rs]15..8 > GPR[rt]15..8) then
  GPR[rd]15..8 ← 1^8
else
  GPR[rd]15..8 ← 0^8
endif

if (GPR[rs]23..16 > GPR[rt]23..16) then
  GPR[rd]23..16 ← 1^8
else
  GPR[rd]23..16 ← 0^8
endif

if (GPR[rs]31..24 > GPR[rt]31..24) then
  GPR[rd]31..24 ← 1^8
else
  GPR[rd]31..24 ← 0^8
endif
```

```
if (GPR[rs]39..32 > GPR[rt]39..32) then
  GPR[rd]39..32 ← 1^8
else
  GPR[rd]39..32 ← 0^8
endif

if (GPR[rs]47..40 > GPR[rt]47..40) then
  GPR[rd]47..40 ← 1^8
else
  GPR[rd]47..40 ← 0^8
endif

if (GPR[rs]55..48 > GPR[rt]55..48) then
  GPR[rd]55..48 ← 1^8
else
  GPR[rd]55..48 ← 0^8
endif

if (GPR[rs]63..56 > GPR[rt]63..56) then
  GPR[rd]63..56 ← 1^8
else
  GPR[rd]63..56 ← 0^8
endif

if (GPR[rs]71..64 > GPR[rt]71..64) then
  GPR[rd]71..64 ← 1^8
else
  GPR[rd]71..64 ← 0^8
endif

if (GPR[rs]79..72 > GPR[rt]79..72) then
  GPR[rd]79..72 ← 1^8
else
  GPR[rd]79..72 ← 0^8
endif

if (GPR[rs]87..80 > GPR[rt]87..80) then
  GPR[rd]87..80 ← 1^8
else
  GPR[rd]87..80 ← 0^8
endif

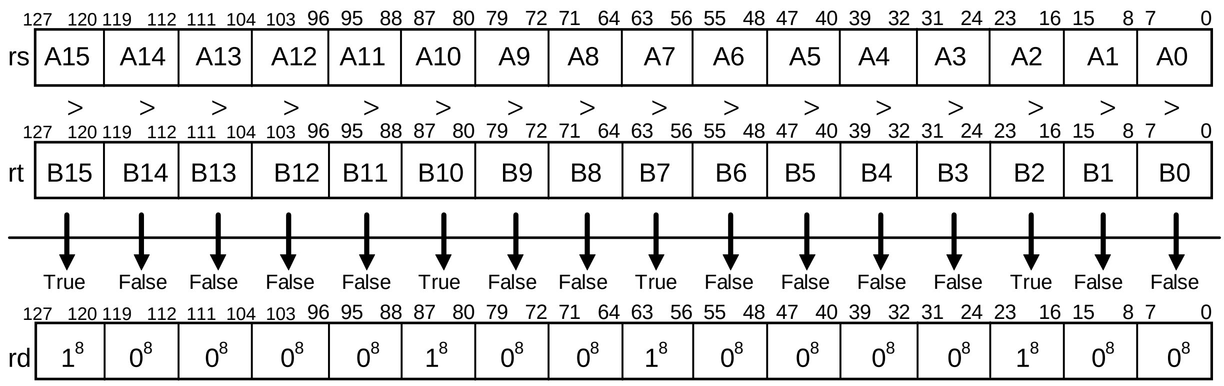
if (GPR[rs]95..88 > GPR[rt]95..88) then
  GPR[rd]95..88 ← 1^8
else
  GPR[rd]95..88 ← 0^8
endif
```

if (GPR[rs]$_{103..96}$ > GPR[rt]$_{103..96}$) then
  GPR[rd]$_{103..96}$ ← 1^8
else
  GPR[rd]$_{103..96}$ ← 0^8
endif

if (GPR[rs]$_{111..104}$ > GPR[rt]$_{111..104}$) then
  GPR[rd]$_{111..104}$ ← 1^8
else
  GPR[rd]$_{111..104}$ ← 0^8
endif

if (GPR[rs]$_{119..112}$ > GPR[rt]$_{119..112}$) then
  GPR[rd]$_{119..112}$ ← 1^8
else
  GPR[rd]$_{119..112}$ ← 0^8
endif

if (GPR[rs]$_{127..120}$ > GPR[rt]$_{127..120}$) then
  GPR[rd]$_{127..120}$ ← 1^8
else
  GPR[rd]$_{127..120}$ ← 0^8
endif



Exceptions:

None

PCGTH Parallel Compare for Greater Than Halfword PCGTH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PCGTH 00110	MMI0 001000
6	5	5	5	5	6

C790

Format: PCGTH rd, rs, rt

Purpose: To record the results of 8 greater-than comparisons in parallel.

Description: rd ← (rs > rt)

The eight signed halfword values in GPR *rs* are compared to the corresponding eight signed halfword values in GPR *rt* in parallel. The results of the comparison are placed into GPR *rd* as follows:

If the signed halfword value in GPR *rs* is greater than the corresponding signed halfword value in GPR *rt*, then the corresponding halfword in GPR *rd* is set to 0xFFFF otherwise it is set to 0x0000.

This instruction operates on 128-bit registers.

Operation:

```
if (GPR[rs]15..0 > GPR[rt]15..0) then
  GPR[rd]15..0 ← 1^16
else
  GPR[rd]15..0 ← 0^16
endif

if (GPR[rs]31..16 > GPR[rt]31..16) then
  GPR[rd]31..16 ← 1^16
else
  GPR[rd]31..16 ← 0^16
endif

if (GPR[rs]47..32 > GPR[rt]47..32) then
  GPR[rd]47..32 ← 1^16
else
  GPR[rd]47..32 ← 0^16
endif

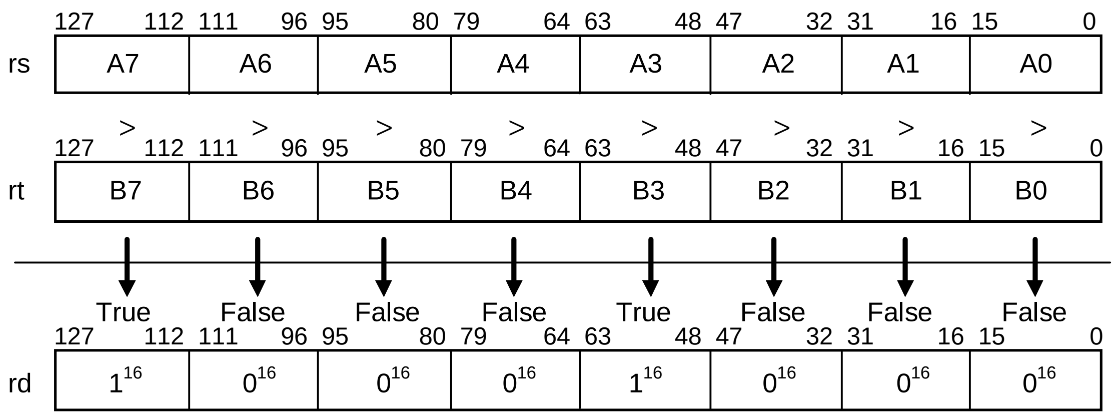
if (GPR[rs]63..48 > GPR[rt]63..48) then
  GPR[rd]63..48 ← 1^16
else
  GPR[rd]63..48 ← 0^16
endif
```

```
if (GPR[rs]79..64 > GPR[rt]79..64) then
  GPR[rd]79..64 ← 1^16
else
  GPR[rd]79..64 ← 0^16
endif

if (GPR[rs]95..80 > GPR[rt]95..80) then
  GPR[rd]95..80 ← 1^16
else
  GPR[rd]95..80 ← 0^16
endif

if (GPR[rs]111..96 > GPR[rt]111..96) then
  GPR[rd]111..96 ← 1^16
else
  GPR[rd]111..96 ← 0^16
endif

if (GPR[rs]127..112 > GPR[rt]127..112) then
  GPR[rd]127..112 ← 1^16
else
  GPR[rd]127..112 ← 0^16
endif
```



Exceptions:

None

PCGTW Parallel Compare for Greater Than Word PCGTW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PCGTW 00010	MMI0 001000
6	5	5	5	5	6

C790

Format: PCGTW rd, rs, rt

Purpose: To record the results of 4 greater-than comparisons in parallel.

Description: rd ← (rs > rt)

The four signed word values in GPR *rs* are compared to the corresponding four signed word values in GPR *rt* in parallel. The results of the comparison are placed into GPR *rd* as follows:

If the signed word value in GPR *rs* is greater than the corresponding signed word value in GPR *rt*, then the corresponding word in GPR *rd* is set 0xFFFFFFFF otherwise it is set to 0x00000000.

This instruction operates on 128-bit registers.

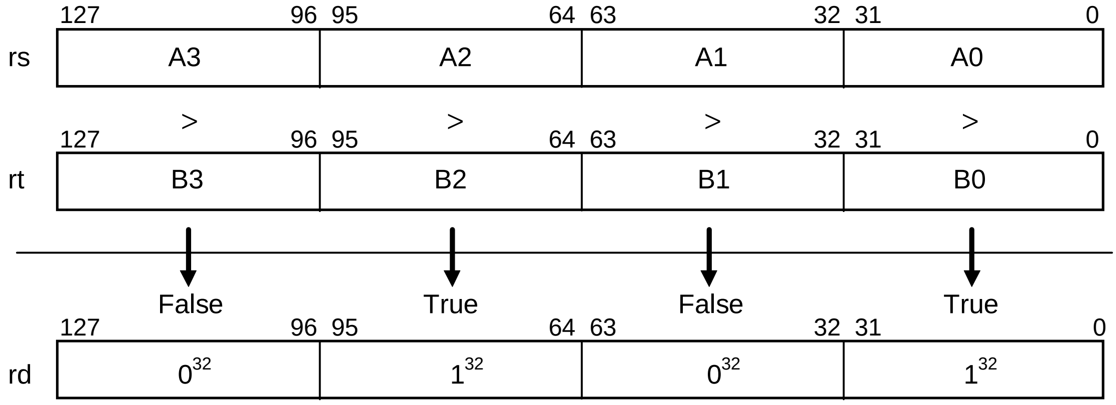
Operation:

if (GPR[rs]$_{31..0}$ > GPR[rt]$_{31..0}$) then
  GPR[rd]$_{31..0}$ ← 1^{32}
else
  GPR[rd]$_{31..0}$ ← 0^{32}
endif

if (GPR[rs]$_{63..32}$ > GPR[rt]$_{63..32}$) then
  GPR[rd]$_{63..32}$ ← 1^{32}
else
  GPR[rd]$_{63..32}$ ← 0^{32}
endif

if (GPR[rs]$_{95..64}$ > GPR[rt]$_{95..64}$) then
  GPR[rd]$_{95..64}$ ← 1^{32}
else
  GPR[rd]$_{95..64}$ ← 0^{32}
endif

if (GPR[rs]$_{127..96}$ > GPR[rt]$_{127..96}$) then
  GPR[rd]$_{127..96}$ ← 1^{32}
else
  GPR[rd]$_{127..96}$ ← 0^{32}
endif



Exception:

None

PCPYH Parallel Copy Halfword PCPYH

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	0 00000	rt	rd	PCPYH 11011	MMI3 101001
6	5	5	5	5	6

C790

Format: PCPYH rd, rt

Purpose: To copy halfword.

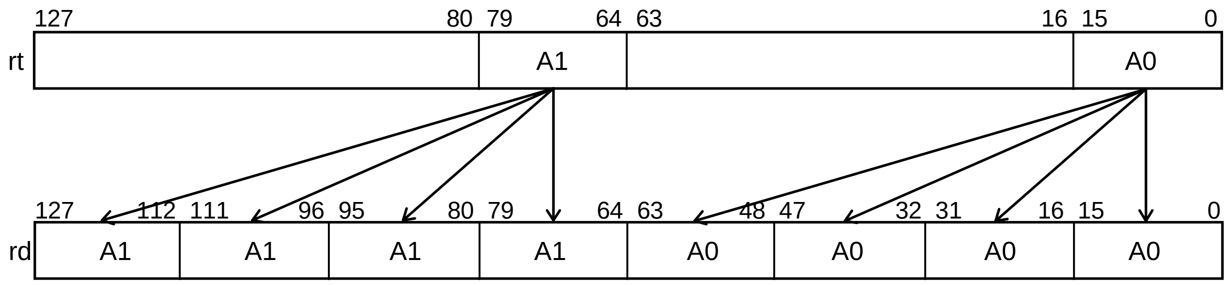
Description: rd ← copy (rt)

The contents of the low-order halfword of the two doublewords in GPR *rt* are copied to each of the halfwords of the two doublewords in GPR *rd*.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{15..0} \leftarrow GPR[rt]_{15..0}$
$GPR[rd]_{31..16} \leftarrow GPR[rt]_{15..0}$
$GPR[rd]_{47..32} \leftarrow GPR[rt]_{15..0}$
$GPR[rd]_{63..48} \leftarrow GPR[rt]_{15..0}$
$GPR[rd]_{79..64} \leftarrow GPR[rt]_{79..64}$
$GPR[rd]_{95..80} \leftarrow GPR[rt]_{79..64}$
$GPR[rd]_{111..96} \leftarrow GPR[rt]_{79..64}$
$GPR[rd]_{127..112} \leftarrow GPR[rt]_{79..64}$



Exceptions:

None

PCPYLD Parallel Copy Lower Doubleword PCPYLD

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PCPYLD 01110	MMI2 001001
6	5	5	5	5	6

C790

Format: PCPYLD rd, rs, rt

Purpose: To copy doubleword.

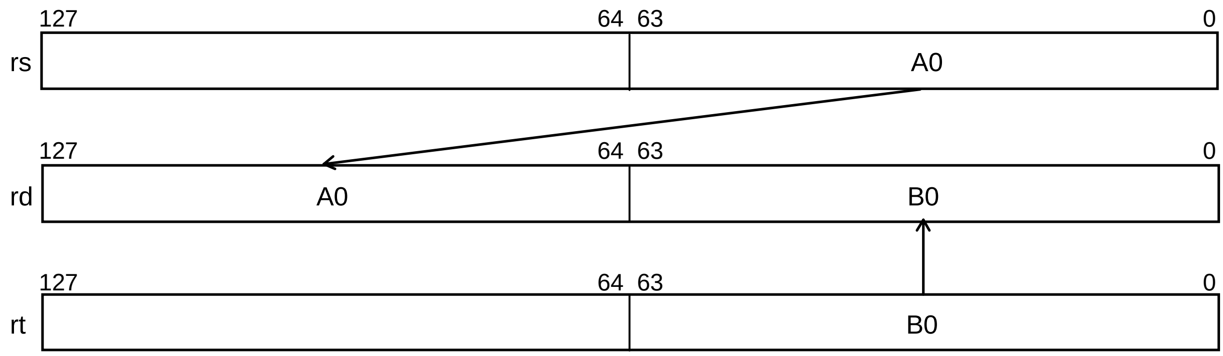
Description: rd ← copy (rs, rt)

The contents of the low-order doubleword in GPR *rs* are combined with the contents of the low-order doubleword in GPR *rt*. The quadword result is placed into GPR *rd*.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{63..0} \leftarrow GPR[rt]_{63..0}$
$GPR[rd]_{127..64} \leftarrow GPR[rs]_{63..0}$



Exceptions:

None

PCPYUD   Parallel Copy Upper Doubleword   PCPYUD

31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 6	5 ... 0
MMI 011100	rs	rt	rd	PCPYUD 01110	MMI3 101001
6	5	5	5	5	6

C790

Format: PCPYUD rd, rs, rt

Purpose: To copy doubleword.

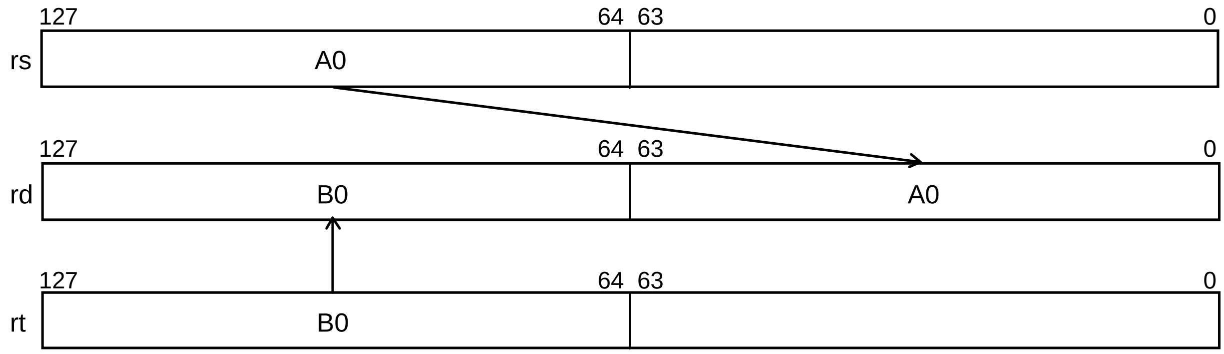
Description: rd ← copy (rs, rt)

The contents of the high-order doubleword in GPR *rs* are combined with the contents of the high-order doubleword in GPR *rt*. The quadword result is placed into GPR *rd*.

This instruction operates on 128-bit registers.

Operation

$GPR[rd]_{63..0} \leftarrow GPR[rs]_{127..64}$
$GPR[rd]_{127..64} \leftarrow GPR[rt]_{127..64}$



Exceptions:

None

PDIVBW

Parallel Divide Broadcast Word

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	0 00000	PDIVBW 11101	MMI2 001001
6	5	5	5	5	6

C790

Format: PDIVBW rs, rt

Purpose: To divide 4 32-bit signed integers by a 16-bit signed integer in parallel.

Description: (LO, HI) ← rs / rt

The four signed words in GPR *rs* are divided by the low-order signed halfword in GPR *rt*, in parallel. The four 32-bit quotients are placed into special register *LO*. The four 16-bit remainders are placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

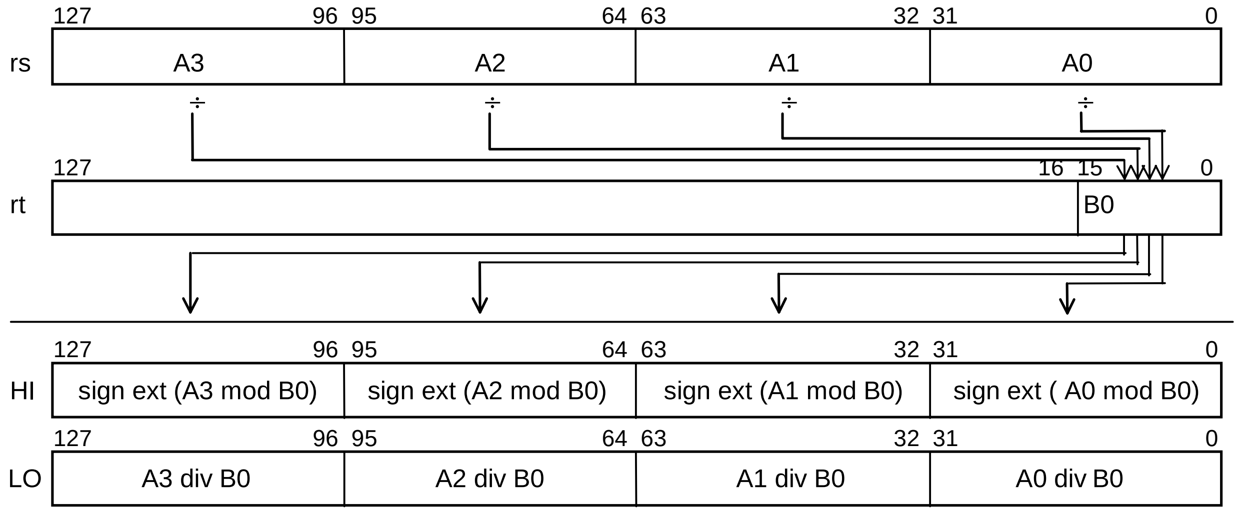
This instruction operates on 128-bit registers.

Restrictions:

If the divisor in GPR *rt* is zero, the arithmetic result value is undefined.

Operation:

$q0 \leftarrow GPR[rs]_{31..0}$ div $GPR[rt]_{15..0}$
$r0 \leftarrow GPR[rs]_{31..0}$ mod $GPR[rt]_{15..0}$
$q1 \leftarrow GPR[rs]_{63..32}$ div $GPR[rt]_{15..0}$
$r1 \leftarrow GPR[rs]_{63..32}$ mod $GPR[rt]_{15..0}$
$q2 \leftarrow GPR[rs]_{95..64}$ div $GPR[rt]_{15..0}$
$r2 \leftarrow GPR[rs]_{95..64}$ mod $GPR[rt]_{15..0}$
$q3 \leftarrow GPR[rs]_{127..96}$ div $GPR[rt]_{15..0}$
$r3 \leftarrow GPR[rs]_{127..96}$ mod $GPR[rt]_{15..0}$
$LO_{31..0} \leftarrow q0_{31..0}$
$HI_{31..0} \leftarrow (r0_{15})^{16} \| r0_{15..0}$
$LO_{63..32} \leftarrow q1_{31..0}$
$HI_{63..32} \leftarrow (r1_{15})^{16} \| r1_{15..0}$
$LO_{95..64} \leftarrow q2_{31..0}$
$HI_{95..64} \leftarrow (r2_{15})^{16} \| r2_{15..0}$
$LO_{127..96} \leftarrow q3_{31..0}$
$HI_{127..96} \leftarrow (r3_{15})^{16} \| r3_{15..0}$



Supplementary explanation:

When 0x80000000 (-2147483648), the most negative value, is divided by 0xFFFF (-1), the operation will results in an overflow. However, overflow exception doesn't occur and the operation results in the following:

Quotient is 0x80000000 (-2147483648), and remainder is 0x00000000 (0).

Exceptions:

None

Programming Notes:

In the C790 the integer divide operation proceeds asynchronously and allows other CPU instructions to execute before it is retired. An attempt to read *LO* or *HI* before the results are written will cause an interlock until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the divide so that other instructions can execute in parallel.

No arithmetic exception occurs under any circumstances. If divide-by-zero or overflow conditions should be detected and some action taken, then the divide instruction is typically followed by additional instructions to check for a zero divisor and / or for overflow. If the divide is asynchronous then the zero-divisor check can execute in parallel with the divide. The action taken on either divide-by-zero or overflow is either a convention within the program itself or more typically, the system software; one possibility is to take a BREAK exception with a code field value to signal the problem to the system software.

As an example, the C programming language in a UNIX environment expects division by zero to either terminate the program or execute a program-specified signal handler. C does not expect overflow to cause any exceptional condition. If the C compiler uses a divide instruction, it also emits code to test for a zero divisor and execute a BREAK instruction to inform the operating system if one is detected.

PDIVUW Parallel Divide Unsigned Word PDIVUW

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	0 00000	PDIVUW 01101	MMI3 101001
6	5	5	5	5	6

C790

Format: PDIVUW rs, rt

Purpose: To divide 2 pairs of 32-bit unsigned integers in parallel.

Description: (LO, HI) ← rs / rt

The low-order unsigned word of the two doublewords in GPR *rs* are divided by the low-order unsigned word of the two doublewords in GPR *rt* in parallel. The two 32 bit quotients are placed into special register *LO*. The two 32-bit remainders are placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

Restrictions:

If neither GPR *rt* nor GPR *rs* contain a zero-extended 32-bit value (bits 127..96 and 63..32 equal zero), the result of the operation will be undefined.

If the divisor in GPR *rt* is zero, the result will be undefined.

Operation:

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif

$q0 \leftarrow (0 \| GPR[rs]_{31..0}) \text{ div } (0 \| GPR[rt]_{31..0})$
$r0 \leftarrow (0 \| GPR[rs]_{31..0}) \text{ mod } (0 \| GPR[rt]_{31..0})$
$q1 \leftarrow (0 \| GPR[rs]_{95..64}) \text{ div } (0 \| GPR[rt]_{95..64})$
$r1 \leftarrow (0 \| GPR[rs]_{95..64}) \text{ mod } (0 \| GPR[rt]_{95..64})$
$LO_{63..0} \leftarrow (q0_{31})^{32} \| q0_{31..0}$
$HI_{63..0} \leftarrow (r0_{31})^{32} \| r0_{31..0}$
$LO_{127..64} \leftarrow (q1_{31})^{32} \| q1_{31..0}$
$HI_{127..64} \leftarrow (r1_{31})^{32} \| r1_{31..0}$

	127–96	95–64	63–32	31–0
rs		A1		A0
		÷		÷
rt		B1		B0
HI	sign ext	(0 ‖ A1) mod (0 ‖ B1)	sign ext	(0 ‖ A0) mod (0 ‖ B0)
LO	sign ext	(0 ‖ A1) div (0 ‖ B1)	sign ext	(0 ‖ A0) div (0 ‖ B0)

Exceptions:

None

Programming Notes:

See the Programming Notes for the PDIVBW instruction.

PDIVW Parallel Divide Word PDIVW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	0 00000	PDIVW 01101	MMI2 001001
6	5	5	5	5	6

C790

Format: PDIVW rs, rt

Purpose: To divide 2 pairs of 32-bit signed integers in parallel.

Description: (LO, HI) ← rs / rt

The low-order signed word of the two doublewords in GPR *rs* are divided by the low-order signed word of the two doublewords in GPR *rt* in parallel. The two 32 bit quotients are placed into special register *LO*. The two 32-bit remainders are placed into special register *HI*.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

Restrictions:

If neither GPR *rt* nor GPR *rs* contain a sign-extended 32-bit value (bits 127..95 equal and 63..31 equal), the result of the operation will be undefined.

If the divisor in GPR *rt* is zero, the result will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif

q0 ← $GPR[rs]_{31..0}$ div $GPR[rt]_{31..0}$
r0 ← $GPR[rs]_{31..0}$ mod $GPR[rt]_{31..0}$
q1 ← $GPR[rs]_{95..64}$ div $GPR[rt]_{95..64}$
r1 ← $GPR[rs]_{95..64}$ mod $GPR[rt]_{95..64}$
$LO_{63..0}$ ← $(q0_{31})^{32} \| q0_{31..0}$
$HI_{63..0}$ ← $(r0_{31})^{32} \| r0_{31..0}$
$LO_{127..64}$ ← $(q1_{31})^{32} \| q1_{31..0}$
$HI_{127..64}$ ← $(r1_{31})^{32} \| r1_{31..0}$

	127 96	95 64	63 32	31 0
rs		A1		A0
		÷		÷
rt		B1		B0

	127 96	95 64	63 32	31 0
HI	sign ext	A1 mod B1	sign ext	A0 mod B0
LO	sign ext	A1 div B1	sign ext	A0 div B0

Supplementary explanation:

When 0x80000000 (-2147483648), the most negative value, is divided by 0xFFFFFFFF (-1), the operation results in an overflow. However, overflow exception doesn't occur; the operation results in the followings:

Quotient (q) is 0x80000000 (-2147483648), and remainder (r) is 0x00000000(0).

Exceptions:

None

Programming Notes:

See the Programming Notes for the PDIVBW instruction.

PEXCH Parallel Exchange Center Halfword PEXCH

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	0 00000	rt	rd	PEXCH 11010	MMI3 101001
6	5	5	5	5	6

C790

Format: PEXCH rd, rt

Purpose: To exchange halfwords.

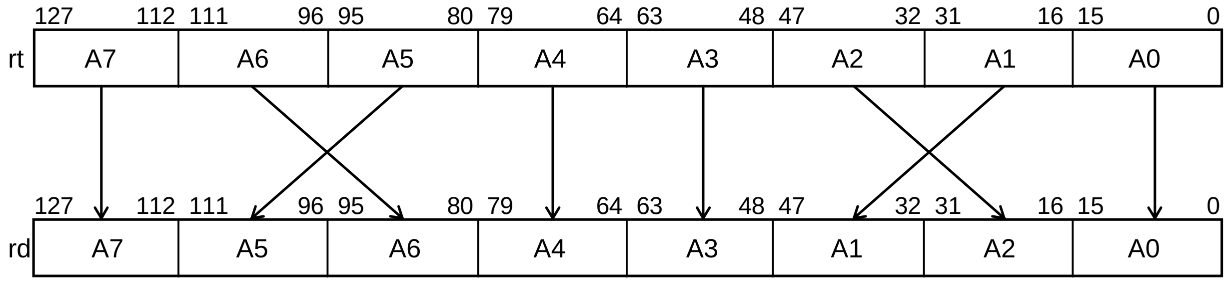
Description: rd ← exchange (rt)

The two central halfwords of the high-order doubleword in GPR *rt* are exchanged and the two central halfwords of the low-order doubleword in GPR *rt* are exchanged. The results are copied to GPR *rd* while other halfwords are copied directly to the corresponding halfwords.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{15..0} \leftarrow GPR[rt]_{15..0}$
$GPR[rd]_{31..16} \leftarrow GPR[rt]_{47..32}$
$GPR[rd]_{47..32} \leftarrow GPR[rt]_{31..16}$
$GPR[rd]_{63..48} \leftarrow GPR[rt]_{63..48}$
$GPR[rd]_{79..64} \leftarrow GPR[rt]_{79..64}$
$GPR[rd]_{95..80} \leftarrow GPR[rt]_{111..96}$
$GPR[rd]_{111..96} \leftarrow GPR[rt]_{95..80}$
$GPR[rd]_{127..112} \leftarrow GPR[rt]_{127..112}$



Exceptions:

None

PEXCW Parallel Exchange Center Word PEXCW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	0 00000	rt	rd	PEXCW 11110	MMI3 101001
6	5	5	5	5	6

C790

Format: PEXCW rd, rt

Purpose: To exchange words.

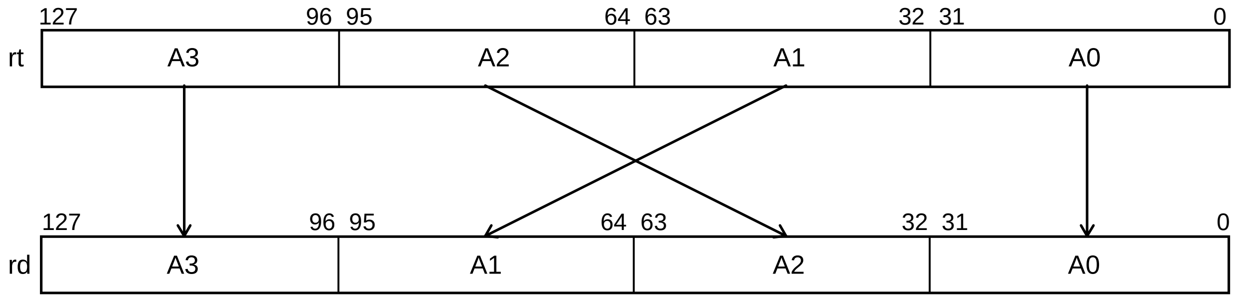
Description: rd ← exchange (rt)

The two central words in GPR *rt* are exchanged. The results are copied to GPR *rd* while other words are copied directly to the corresponding words.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{31..0} \leftarrow GPR[rt]_{31..0}$
$GPR[rd]_{63..32} \leftarrow GPR[rt]_{95..64}$
$GPR[rd]_{95..64} \leftarrow GPR[rt]_{63..32}$
$GPR[rd]_{127..96} \leftarrow GPR[rt]_{127..96}$



Exceptions:

None

PEXEH Parallel Exchange Even Halfword PEXEH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	0 00000	rt	rd	PEXEH 11010	MMI2 001001
6	5	5	5	5	6

C790

Format: PEXEH rd, rt

Purpose: To exchange halfwords.

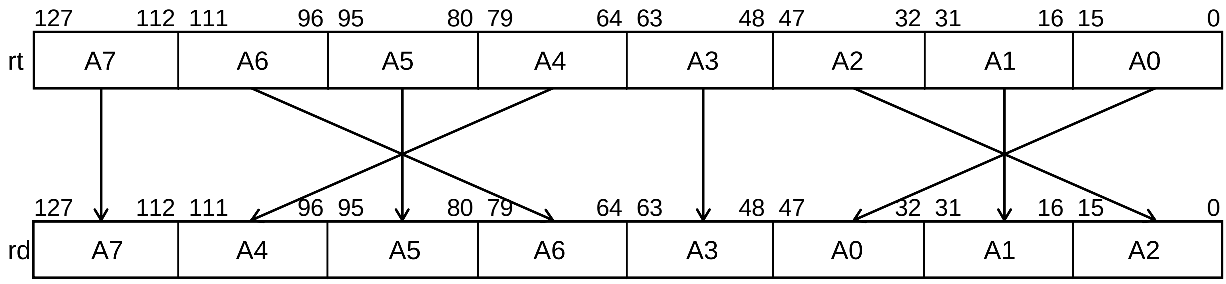
Description: rd ← exchange (rt)

The two low-order halfwords of the two words of the high-order doubleword in GPR *rt* are exchanged and the two low-order halfwords of the two words of the low-order doubleword in GPR *rt* are exchanged. The results are copied to GPR *rd* while other halfwords are copied directly to the corresponding halfwords.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{15..0} \leftarrow GPR[rt]_{47..32}$
$GPR[rd]_{31..16} \leftarrow GPR[rt]_{31..16}$
$GPR[rd]_{47..32} \leftarrow GPR[rt]_{15..0}$
$GPR[rd]_{63..48} \leftarrow GPR[rt]_{63..48}$
$GPR[rd]_{79..64} \leftarrow GPR[rt]_{111..96}$
$GPR[rd]_{95..80} \leftarrow GPR[rt]_{95..80}$
$GPR[rd]_{111..96} \leftarrow GPR[rt]_{79..64}$
$GPR[rd]_{127..112} \leftarrow GPR[rt]_{127..112}$



Exceptions:

None

PEXEW

Parallel Exchange Even Word

PEXEW

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	0 00000	rt	rd	PEXEW 11110	MMI2 001001
6	5	5	5	5	6

C790

Format: PEXEW rd, rt

Purpose: To exchange word.

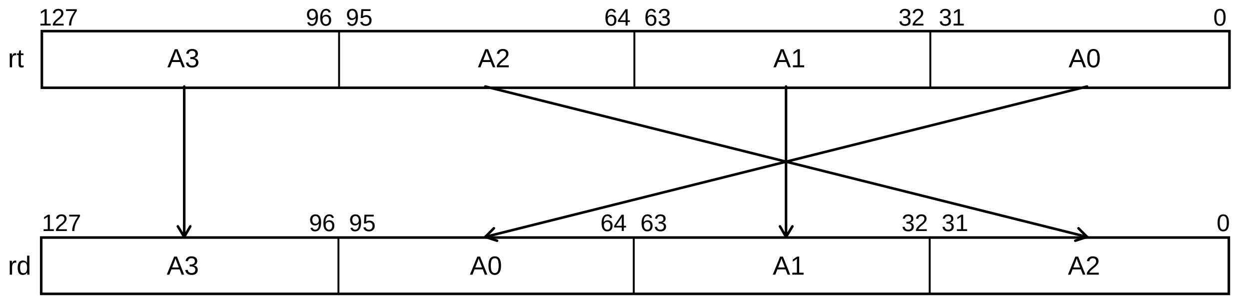
Description: rd ← exchange (rt)

The two low-order words of the two doublewords in GPR *rt* are exchanged. The results are copied to GPR *rd* while other words are copied directly to the corresponding words.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{31..0} \leftarrow GPR[rt]_{95..64}$
$GPR[rd]_{63..32} \leftarrow GPR[rt]_{63..32}$
$GPR[rd]_{95..64} \leftarrow GPR[rt]_{31..0}$
$GPR[rd]_{127..96} \leftarrow GPR[rt]_{127..96}$



Exceptions:

None

PEXT5 Parallel Extend from 5-bits PEXT5

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	0 00000	rt	rd	PEXT5 11110	MMI0 001000
6	5	5	5	5	6

C790

Format: PEXT5 rd, rt

Purpose: To extend bytes from 5-bits.

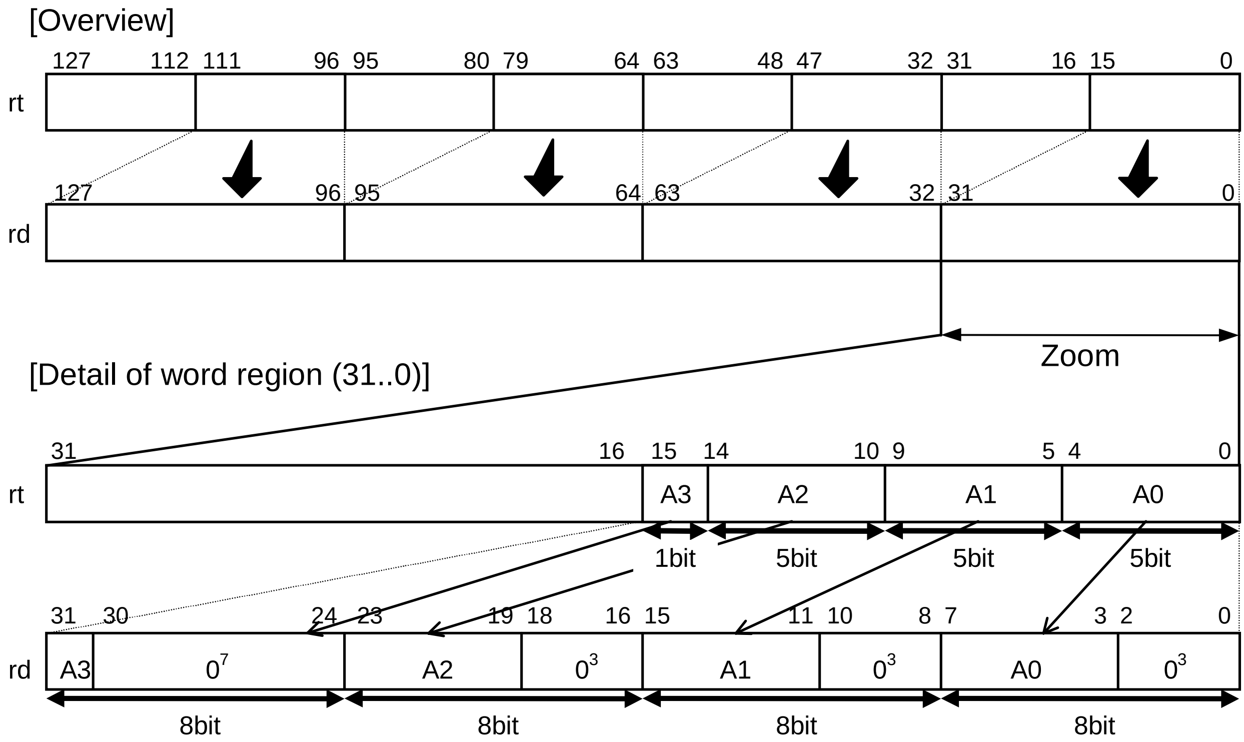
Description: rd ← extend (rt)

The four low-order 16-bits (1, 5, 5, 5 bit) of the four words in GPR *rt* are extended to four 32-bits (8, 8, 8, 8 bit). The quadword result is placed into GPR *rd*.

This instruction operates on 128-bit registers.

Operation

$GPR[rd]_{2..0} \leftarrow 0^3$
$GPR[rd]_{7..3} \leftarrow GPR[rt]_{4..0}$
$GPR[rd]_{10..8} \leftarrow 0^3$
$GPR[rd]_{15..11} \leftarrow GPR[rt]_{9..5}$
$GPR[rd]_{18..16} \leftarrow 0^3$
$GPR[rd]_{23..19} \leftarrow GPR[rt]_{14..10}$
$GPR[rd]_{30..24} \leftarrow 0^7$
$GPR[rd]_{31} \leftarrow GPR[rt]_{15}$
$GPR[rd]_{34..32} \leftarrow 0^3$
$GPR[rd]_{39..35} \leftarrow GPR[rt]_{36..32}$
$GPR[rd]_{42..40} \leftarrow 0^3$
$GPR[rd]_{47..43} \leftarrow GPR[rt]_{41..37}$
$GPR[rd]_{50..48} \leftarrow 0^3$
$GPR[rd]_{55..51} \leftarrow GPR[rt]_{46..42}$
$GPR[rd]_{62..56} \leftarrow 0^7$
$GPR[rd]_{63} \leftarrow GPR[rt]_{47}$
$GPR[rd]_{66..64} \leftarrow 0^3$
$GPR[rd]_{71..67} \leftarrow GPR[rt]_{68..64}$
$GPR[rd]_{74..72} \leftarrow 0^3$
$GPR[rd]_{79..75} \leftarrow GPR[rt]_{73..69}$
$GPR[rd]_{82..80} \leftarrow 0^3$
$GPR[rd]_{87..83} \leftarrow GPR[rt]_{78..74}$
$GPR[rd]_{94..88} \leftarrow 0^7$
$GPR[rd]_{95} \leftarrow GPR[rt]_{79}$
$GPR[rd]_{98..96} \leftarrow 0^3$
$GPR[rd]_{103..99} \leftarrow GPR[rt]_{100..96}$
$GPR[rd]_{106..104} \leftarrow 0^3$
$GPR[rd]_{111..107} \leftarrow GPR[rt]_{105..101}$
$GPR[rd]_{114..112} \leftarrow 0^3$
$GPR[rd]_{119..115} \leftarrow GPR[rt]_{110..106}$
$GPR[rd]_{126..120} \leftarrow 0^7$
$GPR[rd]_{127} \leftarrow GPR[rt]_{111}$



Exceptions:

None

PEXTLB Parallel Extend Lower from Byte PEXTLB

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PEXTLB 11010	MMI0 001000
6	5	5	5	5	6

C790

Format: PEXTLB rd, rs, rt

Purpose: To extend halfwords from bytes.

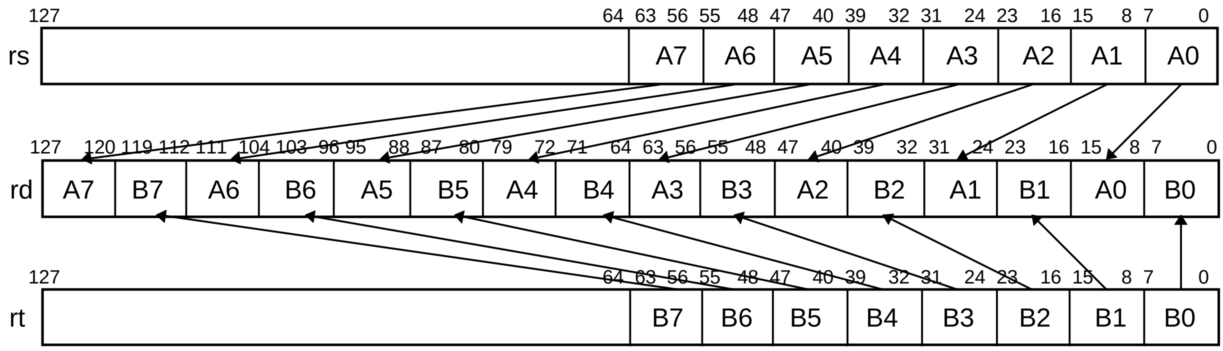
Description: rd ← extend (rs, rt)

The contents of the low-order doubleword in GPR *rs* are combined with the contents of the low-order doubleword in GPR *rt* in a byte wide Interleaved operation. The quadword result is placed into GPR *rd.*

This instruction operates on 128-bit registers.

Operation

$GPR[rd]_{7..0} \leftarrow GPR[rt]_{7..0}$
$GPR[rd]_{15..8} \leftarrow GPR[rs]_{7..0}$
$GPR[rd]_{23..16} \leftarrow GPR[rt]_{15..8}$
$GPR[rd]_{31..24} \leftarrow GPR[rs]_{15..8}$
$GPR[rd]_{39..32} \leftarrow GPR[rt]_{23..16}$
$GPR[rd]_{47..40} \leftarrow GPR[rs]_{23..16}$
$GPR[rd]_{55..48} \leftarrow GPR[rt]_{31..24}$
$GPR[rd]_{63..56} \leftarrow GPR[rs]_{31..24}$
$GPR[rd]_{71..64} \leftarrow GPR[rt]_{39..32}$
$GPR[rd]_{79..72} \leftarrow GPR[rs]_{39..32}$
$GPR[rd]_{87..80} \leftarrow GPR[rt]_{47..40}$
$GPR[rd]_{95..88} \leftarrow GPR[rs]_{47..40}$
$GPR[rd]_{103..96} \leftarrow GPR[rt]_{55..48}$
$GPR[rd]_{111..104} \leftarrow GPR[rs]_{55..48}$
$GPR[rd]_{119..112} \leftarrow GPR[rt]_{63..56}$
$GPR[rd]_{127..120} \leftarrow GPR[rs]_{63..56}$



Exceptions:

None

PEXTLH — Parallel Extend Lower from Halfword — PEXTLH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PEXTLH 10110	MMI0 001000
6	5	5	5	5	6

C790

Format: PEXTLH rd, rs, rt

Purpose: To extend words from halfwords.

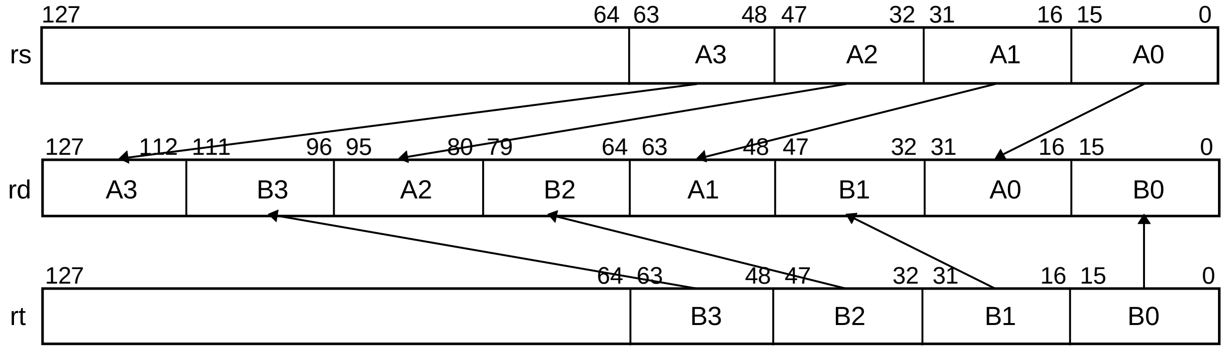
Description: rd ← extend (rs, rt)

The contents of the low-order doubleword in GPR *rs* are combined with the contents of the low-order doubleword in GPR *rt* in a halfword wide Interleaved operation. The quadword result is placed into GPR *rd*.

This instruction operates on 128-bit registers.

Operation

$GPR[rd]_{15..0} \leftarrow GPR[rt]_{15..0}$
$GPR[rd]_{31..16} \leftarrow GPR[rs]_{15..0}$
$GPR[rd]_{47..32} \leftarrow GPR[rt]_{31..16}$
$GPR[rd]_{63..48} \leftarrow GPR[rs]_{31..16}$
$GPR[rd]_{79..64} \leftarrow GPR[rt]_{47..32}$
$GPR[rd]_{95..80} \leftarrow GPR[rs]_{47..32}$
$GPR[rd]_{111..96} \leftarrow GPR[rt]_{63..48}$
$GPR[rd]_{127..112} \leftarrow GPR[rs]_{63..48}$



Exceptions:

None

PEXTLW Parallel Extend Lower from Word PEXTLW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PEXTLW 10010	MMI0 001000
6	5	5	5	5	6

C790

Format: PEXTLW rd, rs, rt

Purpose: To extend doublewords from words.

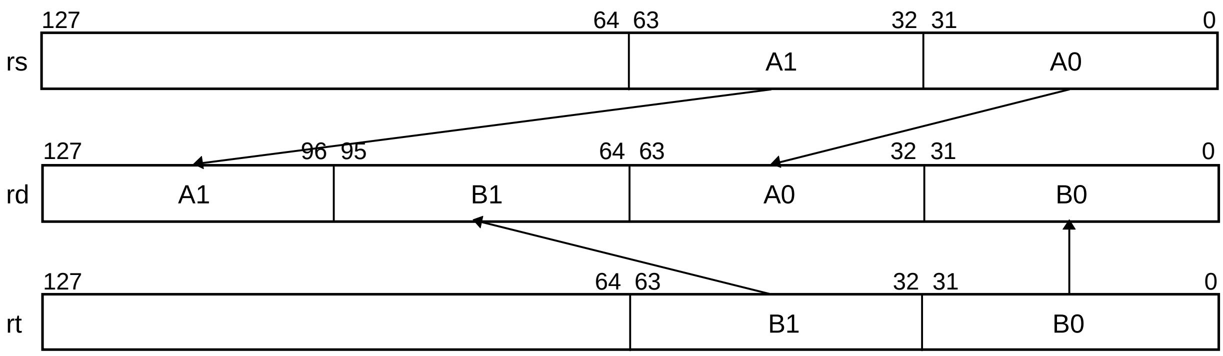
Description: rd ← extend (rs, rt)

The contents of the low-order doubleword in GPR *rs* are combined with the contents of the low-order doubleword in GPR *rt* in a word wide Interleaved operation. The quadword result is placed into GPR *rd*.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{31..0} \leftarrow GPR[rt]_{31..0}$
$GPR[rd]_{63..32} \leftarrow GPR[rs]_{31..0}$
$GPR[rd]_{95..64} \leftarrow GPR[rt]_{63..32}$
$GPR[rd]_{127..96} \leftarrow GPR[rs]_{63..32}$



Exceptions:

None

PEXTUB

Parallel Extend Upper from Byte

PEXTUB

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PEXTUB 11010	MMI1 101000
6	5	5	5	5	6

C790

Format: PEXTUB rd, rs, rt

Purpose: To extend halfwords from bytes.

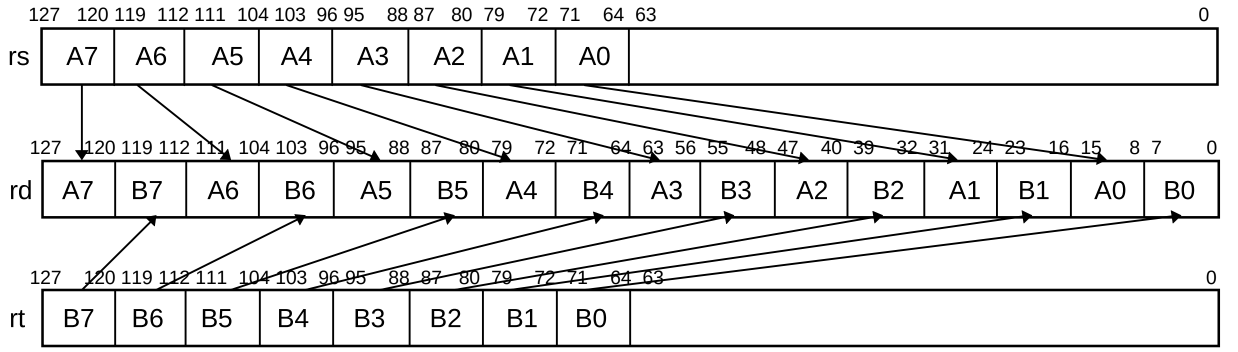
Description: rd ← extend (rs, rt)

The contents of the high-order doubleword in GPR *rs* are combined with the contents of the high-order doubleword in GPR *rt* in a byte wide Interleaved operation. The quadword result is placed into GPR *rd.*

This instruction operates on 128-bit registers.

Operation:

GPR[rd]$_{7..0}$ ← GPR[rt]$_{71..64}$
GPR[rd]$_{15..8}$ ← GPR[rs]$_{71..64}$
GPR[rd]$_{23..16}$ ← GPR[rt]$_{79..72}$
GPR[rd]$_{31..24}$ ← GPR[rs]$_{79..72}$
GPR[rd]$_{39..32}$ ← GPR[rt]$_{87..80}$
GPR[rd]$_{47..40}$ ← GPR[rs]$_{87..80}$
GPR[rd]$_{55..48}$ ← GPR[rt]$_{95..88}$
GPR[rd]$_{63..56}$ ← GPR[rs]$_{95..88}$
GPR[rd]$_{71..64}$ ← GPR[rt]$_{103..96}$
GPR[rd]$_{79..72}$ ← GPR[rs]$_{103..96}$
GPR[rd]$_{87..80}$ ← GPR[rt]$_{111..104}$
GPR[rd]$_{95..88}$ ← GPR[rs]$_{111..104}$
GPR[rd]$_{103..96}$ ← GPR[rt]$_{119..112}$
GPR[rd]$_{111..104}$ ← GPR[rs]$_{119..112}$
GPR[rd]$_{119..112}$ ← GPR[rt]$_{127..120}$
GPR[rd]$_{127..120}$ ← GPR[rs]$_{127..120}$



Exceptions:

None

PEXTUH Parallel Extend Upper from Halfword PEXTUH

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PEXTUH 10110	MMI1 101000
6	5	5	5	5	6

C790

Format: PEXTUH rd, rs, rt

Purpose: To extend words from halfwords.

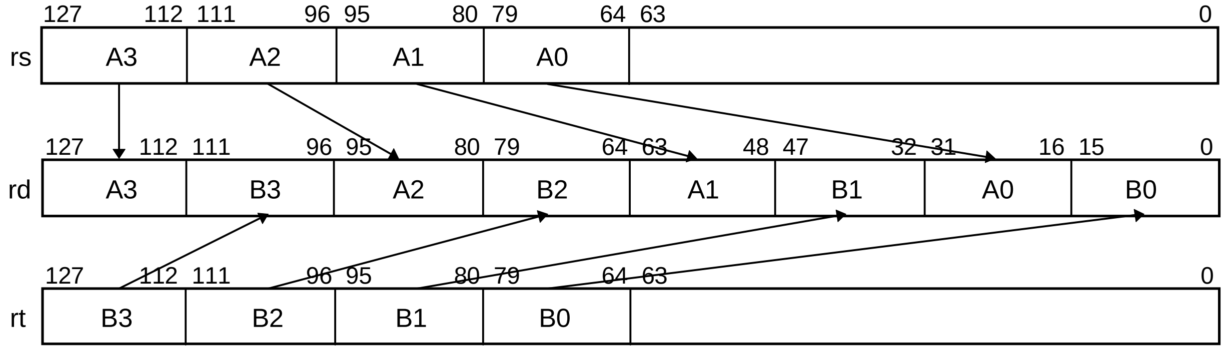
Description: rd ← extend (rs, rt)

The contents of the high-order doubleword in GPR *rs* are combined with the contents of the high-order doubleword in GPR *rt* in a halfword wide Interleaved operation. The quadword result is placed into GPR *rd.*

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{15..0} \leftarrow GPR[rt]_{79..64}$
$GPR[rd]_{31..16} \leftarrow GPR[rs]_{79..64}$
$GPR[rd]_{47..32} \leftarrow GPR[rt]_{95..80}$
$GPR[rd]_{63..48} \leftarrow GPR[rs]_{95..80}$
$GPR[rd]_{79..64} \leftarrow GPR[rt]_{111..96}$
$GPR[rd]_{95..80} \leftarrow GPR[rs]_{111..96}$
$GPR[rd]_{111..96} \leftarrow GPR[rt]_{127..112}$
$GPR[rd]_{127..112} \leftarrow GPR[rs]_{127..112}$



Exceptions:

None

PEXTUW — Parallel Extend Upper from Word — PEXTUW

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PEXTUW 10010	MMI1 101000
6	5	5	5	5	6

C790

Format: PEXTUW rd, rs, rt

Purpose: To extend doublewords from words.

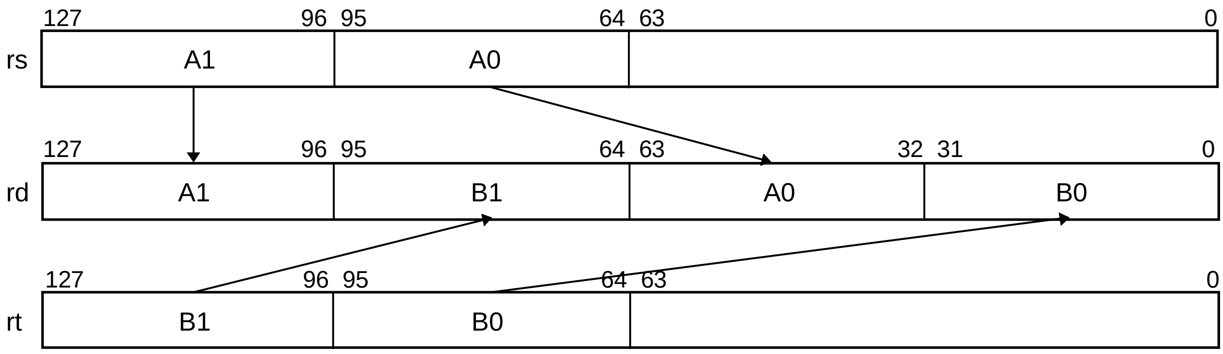
Description: rd ← extend (rs, rt)

The contents of the high-order doubleword in GPR *rs* are combined with the contents of the high-order doubleword in GPR *rt* in a word wide Interleaved operation. The quadword result is placed into GPR *rd.*

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{31..0} \leftarrow GPR[rt]_{95..64}$
$GPR[rd]_{63..32} \leftarrow GPR[rs]_{95..64}$
$GPR[rd]_{95..64} \leftarrow GPR[rt]_{127..96}$
$GPR[rd]_{127..96} \leftarrow GPR[rs]_{127..96}$



Exceptions:

None

PHMADH Parallel Horizontal Multiply-Add Halfword PHMADH

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PHMADH 10001	MMI2 001001
6	5	5	5	5	6

C790

Format: PHMADH rd, rs, rt

Purpose: To multiply 8 pairs of 16-bit signed integers and horizontally add.

Description: (rd, HI, LO) ← rs × rt + rs × rt

The eight signed halfwords in GPR *rs* are multiplied by the eight signed halfwords in GPR *rt* in parallel. The four word multiply results are added to the other four word multiply results, and the four word results are placed into the corresponding words in special registers *HI*, *LO* and GPR *rd.*

No arithmetic exception occurs under any circumstances.

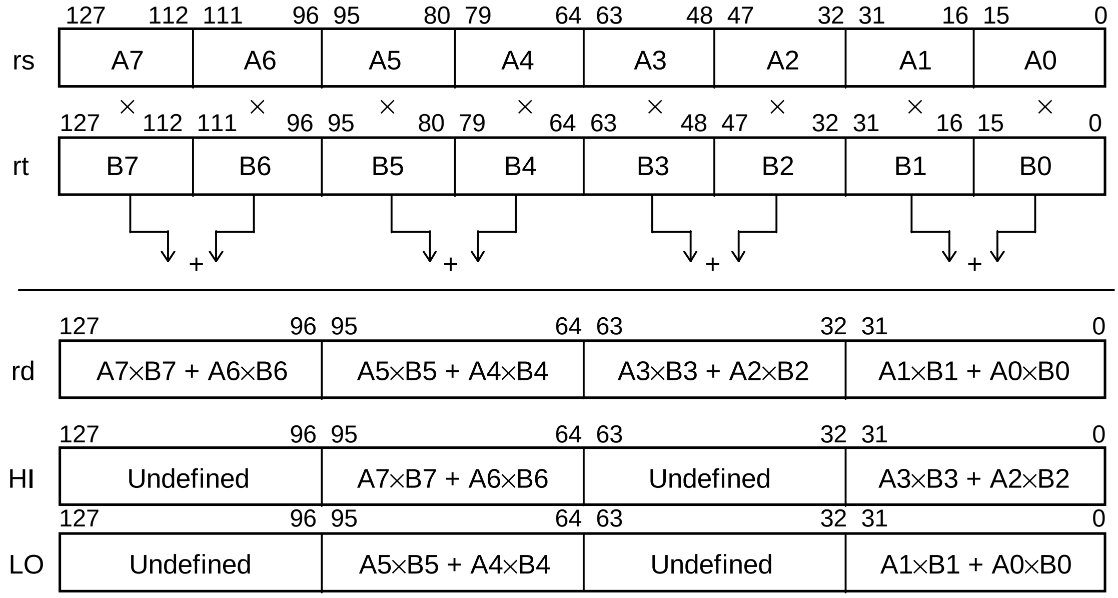
This instruction operates on 128-bit registers.

Restrictions:

None

Operation:

prod0 $\leftarrow$ $GPR[rs]_{31..16} \times GPR[rt]_{31..16} + GPR[rs]_{15..0} \times GPR[rt]_{15..0}$
prod1 $\leftarrow$ $GPR[rs]_{63..48} \times GPR[rt]_{63..48} + GPR[rs]_{47..32} \times GPR[rt]_{47..32}$
prod2 $\leftarrow$ $GPR[rs]_{95..80} \times GPR[rt]_{95..80} + GPR[rs]_{79..64} \times GPR[rt]_{79..64}$
prod3 $\leftarrow$ $GPR[rs]_{127..112} \times GPR[rt]_{127..112} + GPR[rs]_{111..96} \times GPR[rt]_{111..96}$
$LO_{31..0}$ $\leftarrow$ $prod0_{31..0}$
$LO_{63..32}$ $\leftarrow$ Undefined
$HI_{31..0}$ $\leftarrow$ $prod1_{31..0}$
$HI_{63..32}$ $\leftarrow$ Undefined
$LO_{95..64}$ $\leftarrow$ $prod2_{31..0}$
$LO_{127..96}$ $\leftarrow$ Undefined
$HI_{95..64}$ $\leftarrow$ $prod3_{31..0}$
$HI_{127..96}$ $\leftarrow$ Undefined
$GPR[rd]_{31..0}$ $\leftarrow$ $prod0_{31..0}$
$GPR[rd]_{63..32}$ $\leftarrow$ $prod1_{31..0}$
$GPR[rd]_{95..64}$ $\leftarrow$ $prod2_{31..0}$
$GPR[rd]_{127..96}$ $\leftarrow$ $prod3_{31..0}$



Exceptions:

None

Programming Notes:

In the C790, the integer multiply operation allows other CPU instructions to execute out-of-order. An attempt to read *LO* or *HI* registers before the results are written will cause an interlock until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

PHMSBH Parallel Horizontal Multiply-Subtract Halfword PHMSBH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PHMSBH 10101	MMI2 001001
6	5	5	5	5	6

C790

Format: PHMSBH rd, rs, rt

Purpose: To multiply 8 pairs of 16-bit signed integers and horizontally subtract.

Description: (rd, HI, LO) ← rs × rt – rs × rt

The eight signed halfwords in GPR *rs* are multiplied by the eight signed halfwords in GPR *rt* in parallel. The four word multiply results are subtracted from the other four word multiply results, and the four word results are placed into the corresponding words in special registers *HI*, *LO* and GPR *rd*.

No arithmetic exception occurs under any circumstances.

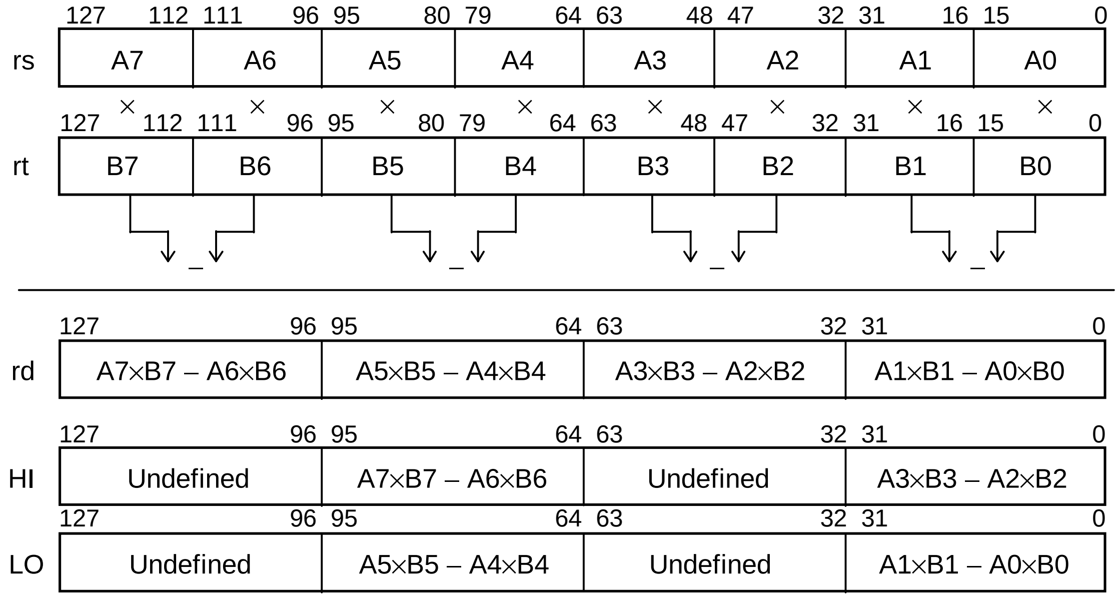
This instruction operates on 128-bit registers.

Restrictions:

None

Operation:

prod0 ← $GPR[rs]_{31..16} \times GPR[rt]_{31..16} - GPR[rs]_{15..0} \times GPR[rt]_{15..0}$
prod1 ← $GPR[rs]_{63..48} \times GPR[rt]_{63..48} - GPR[rs]_{47..32} \times GPR[rt]_{47..32}$
prod2 ← $GPR[rs]_{95..80} \times GPR[rt]_{95..80} - GPR[rs]_{79..64} \times GPR[rt]_{79..64}$
prod3 ← $GPR[rs]_{127..112} \times GPR[rt]_{127..112} - GPR[rs]_{111..96} \times GPR[rt]_{111..96}$
$LO_{31..0}$ ← $prod0_{31..0}$
$LO_{63..32}$ ← Undefined
$HI_{31..0}$ ← $prod1_{31..0}$
$HI_{63..32}$ ← Undefined
$LO_{95..64}$ ← $prod2_{31..0}$
$LO_{127..96}$ ← Undefined
$HI_{95..64}$ ← $prod3_{31..0}$
$HI_{127..96}$ ← Undefined
$GPR[rd]_{31..0}$ ← $prod0_{31..0}$
$GPR[rd]_{63..32}$ ← $prod1_{31..0}$
$GPR[rd]_{95..64}$ ← $prod2_{31..0}$
$GPR[rd]_{127..96}$ ← $prod3_{31..0}$



Exceptions:

None

Programming Notes:

In the C790, the integer multiply operation allows other CPU instructions to execute out-of-order. An attempt to read *LO* or *HI* registers before the results are written will wait (interlock) until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

PINTEH — Parallel Interleave Even Halfword — PINTEH

31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0
MMI 011100	rs	rt	rd	PINTEH 01010	MMI3 101001
6	5	5	5	5	6

C790

Format: PINTEH rd, rs, rt

Purpose: To combine halfwords in a halfword wide interleaved operation.

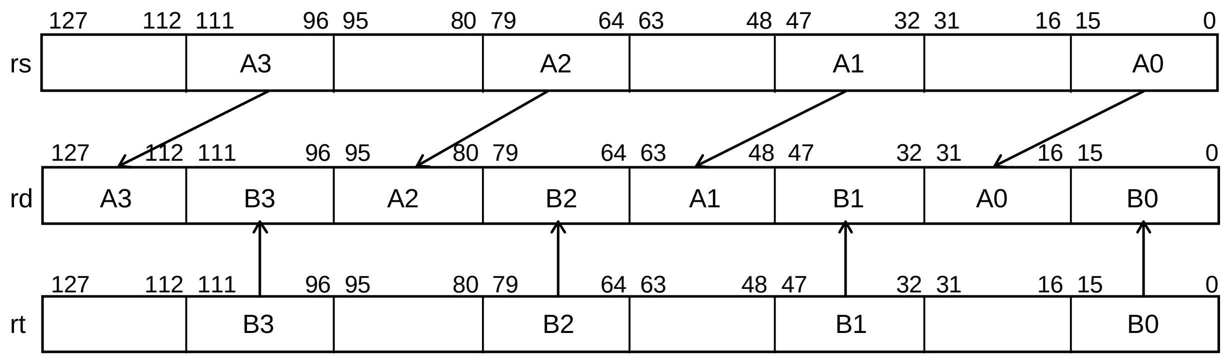
Description: rd ← interleave (rs, rt)

The low-order halfword of the four words in GPR *rs* are combined with the low-order halfword of the four words in GPR *rt* in a halfword wide Interleaved operation. The quadword result is placed into GPR *rd.*

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{15..0} \leftarrow GPR[rt]_{15..0}$
$GPR[rd]_{31..16} \leftarrow GPR[rs]_{15..0}$
$GPR[rd]_{47..32} \leftarrow GPR[rt]_{47..32}$
$GPR[rd]_{63..48} \leftarrow GPR[rs]_{47..32}$
$GPR[rd]_{79..64} \leftarrow GPR[rt]_{79..64}$
$GPR[rd]_{95..80} \leftarrow GPR[rs]_{79..64}$
$GPR[rd]_{111..96} \leftarrow GPR[rt]_{111..96}$
$GPR[rd]_{127..112} \leftarrow GPR[rs]_{111..96}$



Exceptions:

None

PINTH Parallel Interleave Halfword PINTH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PINTH 01010	MMI2 001001
6	5	5	5	5	6

C790

Format: PINTH rd, rs, rt

Purpose: To combine doublewords in a halfword wide interleaved operation.

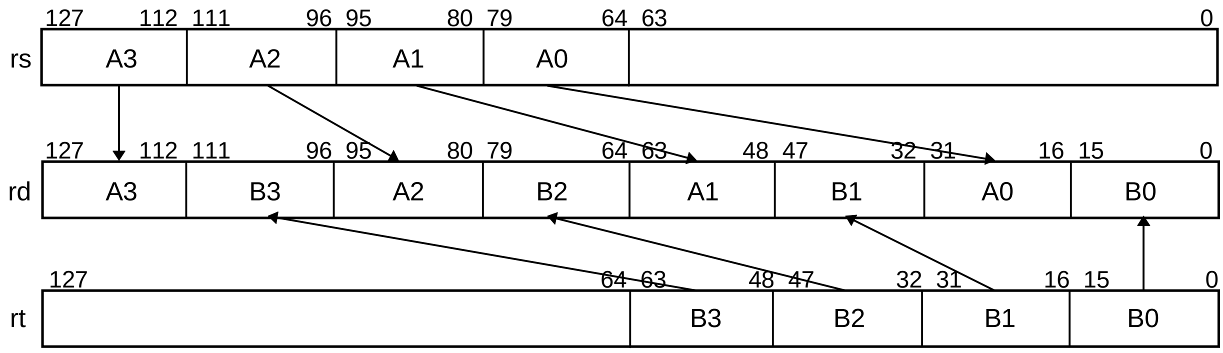
Description: rd ← interleave (rs, rt)

The contents of the high-order doubleword in GPR *rs* are combined with the contents of the low-order doubleword in GPR *rt* in a halfword wide Interleaved operation. The quadword result is placed into GPR *rd.*

This instruction operates on 128-bit registers.

Operation:

GPR[rd]$_{15..0}$ ← GPR[rt]$_{15..0}$
GPR[rd]$_{31..16}$ ← GPR[rs]$_{79..64}$
GPR[rd]$_{47..32}$ ← GPR[rt]$_{31..16}$
GPR[rd]$_{63..48}$ ← GPR[rs]$_{95..80}$
GPR[rd]$_{79..64}$ ← GPR[rt]$_{47..32}$
GPR[rd]$_{95..80}$ ← GPR[rs]$_{111..96}$
GPR[rd]$_{111..96}$ ← GPR[rt]$_{63..48}$
GPR[rd]$_{127..112}$ ← GPR[rs]$_{127..112}$



Exceptions:

None

PLZCW Parallel Leading Zero or one Count Word PLZCW

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	0 00000	rd	0 00000	PLZCW 000100
6	5	5	5	5	6

C790

Format: PLZCW rd, rs

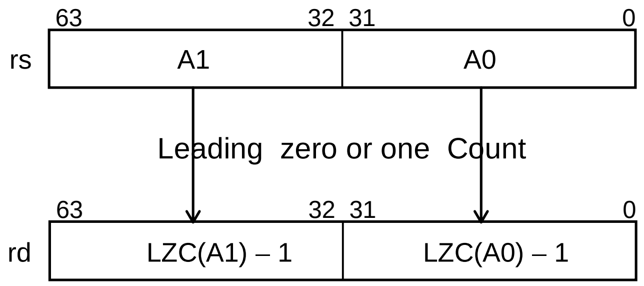
Purpose: To count leading zero (s) or one (s) (2 parallel operations).

Description: rd ← LZC (rs) – 1

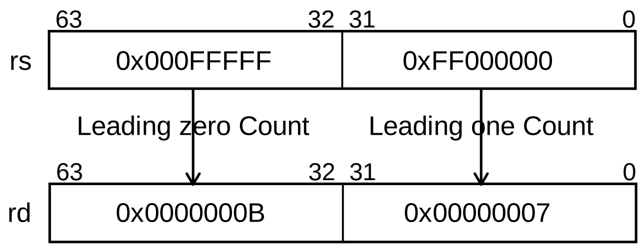
The number of leading zeros or ones of the two words in GPR *rs* are counted. The results of the leading counts minus one are loaded in the corresponding words in GPR *rd*.

Operation:

$GPR[rd]_{31..0}$ ← Leading zero or one count ($GPR[rs]_{31..0}$) – 1
$GPR[rd]_{63..32}$ ← Leading zero or one count ($GPR[rs]_{63..32}$) – 1



Example :



Exceptions:

None

PMADDH

Parallel Multiply-Add Halfword

PMADDH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PMADDH 10000	MMI2 001001
6	5	5	5	5	6

C790

Format: PMADDH rd, rs, rt

Purpose: To multiply 8 pairs of 16-bit signed integers and accumulate, in parallel.

Description: (rd, HI, LO) ← (HI, LO) + rs × rt

The eight signed halfwords in GPR *rs* are multiplied by the eight signed halfwords in GPR *rt* in parallel. The eight word multiply results are added to the corresponding words in special registers *HI* and *LO*, and the word results are placed into the corresponding words in special registers *HI*, *LO* and GPR *rd*.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

Restrictions:

None

Operation:

prod0 ← $LO_{31..0} + GPR[rs]_{15..0} \times GPR[rt]_{15..0}$
prod1 ← $LO_{63..32} + GPR[rs]_{31..16} \times GPR[rt]_{31..16}$
prod2 ← $HI_{31..0} + GPR[rs]_{47..32} \times GPR[rt]_{47..32}$
prod3 ← $HI_{63..32} + GPR[rs]_{63..48} \times GPR[rt]_{63..48}$
prod4 ← $LO_{95..64} + GPR[rs]_{79..64} \times GPR[rt]_{79..64}$
prod5 ← $LO_{127..96} + GPR[rs]_{95..80} \times GPR[rt]_{95..80}$
prod6 ← $HI_{95..64} + GPR[rs]_{111..96} \times GPR[rt]_{111..96}$
prod7 ← $HI_{127..96} + GPR[rs]_{127..112} \times GPR[rt]_{127..112}$
$LO_{31..0}$ ← $prod0_{31..0}$
$LO_{63..32}$ ← $prod1_{31..0}$
$HI_{31..0}$ ← $prod2_{31..0}$
$HI_{63..32}$ ← $prod3_{31..0}$
$LO_{95..64}$ ← $prod4_{31..0}$
$LO_{127..96}$ ← $prod5_{31..0}$
$HI_{95..64}$ ← $prod6_{31..0}$
$HI_{127..96}$ ← $prod7_{31..0}$
$GPR[rd]_{31..0}$ ← $prod0_{31..0}$
$GPR[rd]_{63..32}$ ← $prod2_{31..0}$
$GPR[rd]_{95..64}$ ← $prod4_{31..0}$
$GPR[rd]_{127..96}$ ← $prod6_{31..0}$

	127–112	111–96	95–80	79–64	63–48	47–32	31–16	15–0
rs	A7	A6	A5	A4	A3	A2	A1	A0
	×	×	×	×	×	×	×	×
rt	B7	B6	B5	B4	B3	B2	B1	B0

	127–96	95–64	63–32	31–0
HI	C7	C6	C3	C2
LO	C5	C4	C1	C0

	127–96	95–64	63–32	31–0
rd	A6 × B6 + C6	A4 × B4 + C4	A2 × B2 + C2	A0 × B0 + C0

	127–96	95–64	63–32	31–0
HI	A7 × B7 + C7	A6 × B6 + C6	A3 × B3 + C3	A2 × B2 + C2
LO	A5 × B5 + C5	A4 × B4 + C4	A1 × B1 + C1	A0 × B0 + C0

Exceptions:

None

Programming Notes:

In the C790, the integer multiply operation allow other CPU instructions to execute out-of-order. An attempt to read *LO* or *HI* registers before the results are written will cause an interlock until the results are ready. Asynchronous execution does not affect the program result, but offers an opportunity for performance improvement by scheduling the multiply so that other instructions can execute in parallel.

Programs that require overflow detection must check for it explicitly.

PMADDUW Parallel Multiply-Add Unsigned Word PMADDUW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PMADDUW 00000	MMI3 101001
6	5	5	5	5	6

C790

Format: PMADDUW rd, rs, rt

Purpose: To multiply 2 pairs of 32-bit unsigned integers and accumulate in parallel.

Description: (rd, HI, LO) ← (HI, LO) + rs × rt

The low-order unsigned word of the two doublewords in GPR *rs* are multiplied by the low-order unsigned word of the two doublewords in GPR *rt* in parallel. The two 64-bit multiply results are added to the contents of special registers *HI* and *LO*. The low-order word of the two doubleword results are placed into special register *LO*, and the high-order word of the two doubleword results are placed into special register *HI*. The two doubleword results are placed into GPR *rd*.

No arithmetic exception occurs under any circumstances.

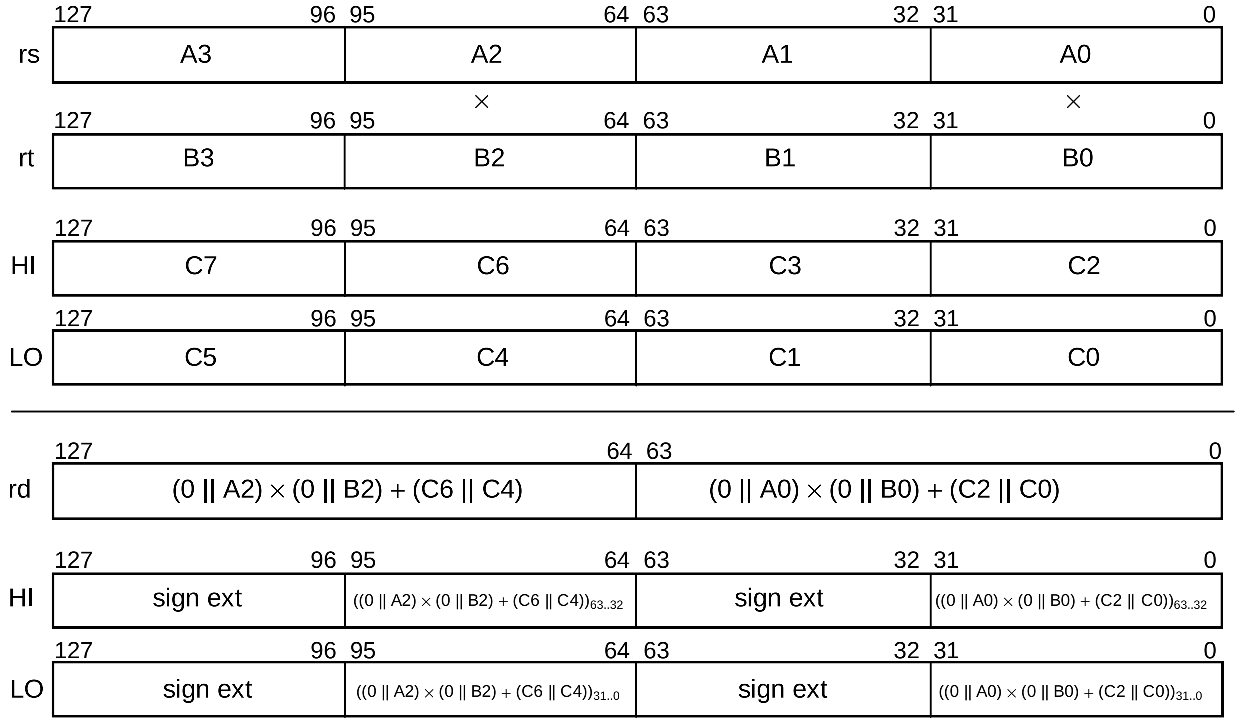
This instruction operates on 128-bit registers.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain zero-extended 32-bit values (bits 127..96 and 63..32 equal zero) then the result of the equation will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod0 ← $(HI_{31..0} \| LO_{31..0}) + (0 \| GPR[rs]_{31..0}) \times (0 \| GPR[rt]_{31..0})$
prod1 ← $(HI_{95..64} \| LO_{95..64}) + (0 \| GPR[rs]_{95..64}) \times (0 \| GPR[rt]_{95..64})$
$LO_{63..0}$ ← $(prod0_{31})^{32} \| prod0_{31..0}$
$HI_{63..0}$ ← $(prod0_{63})^{32} \| prod0_{63..32}$
$LO_{127..64}$ ← $(prod1_{31})^{32} \| prod1_{31..0}$
$HI_{127..64}$ ← $(prod1_{63})^{32} \| prod1_{63..32}$
$GPR[rd]_{63..0}$ ← $prod0_{63..0}$
$GPR[rd]_{127..64}$ ← $prod1_{63..0}$



Exceptions:

None

Programming Notes:

See the Programming Notes for the PMADDH instruction.

PMADDW Parallel Multiply-Add Word PMADDW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PMADDW 00000	MMI2 001001
6	5	5	5	5	6

C790

Format: PMADDW rd, rs, rt

Purpose: To multiply 2 pairs of 32-bit signed integers and accumulate in parallel.

Description: (rd, HI, LO) ← (HI, LO) + rs × rt

The low-order signed word of the two doublewords in GPR *rs* are multiplied by the low-order signed word of the two doublewords in GPR *rt* in parallel. The two 64-bit multiply results are added to the contents of special registers *HI* and *LO*. The low-order word of the two doubleword results are placed into special register *LO*, and the high-order word of the two doubleword results are placed into special register *HI*. The two doubleword results are placed into GPR *rd*.

No arithmetic exception occurs under any circumstances.

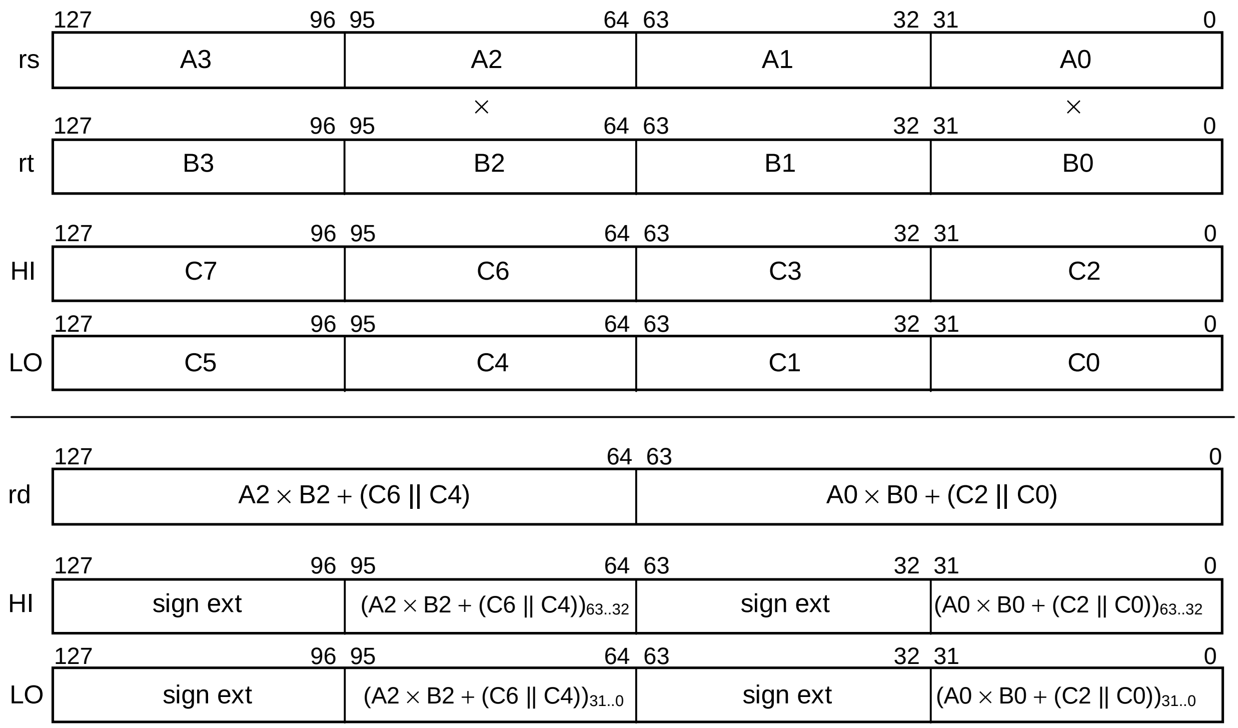
This instruction operates on 128-bit registers.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 127..95 and 63..31 equal) then the result of the equation will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod0 $\leftarrow (HI_{31..0} \| LO_{31..0}) + GPR[rs]_{31..0} \times GPR[rt]_{31..0}$
prod1 $\leftarrow (HI_{95..64} \| LO_{95..64}) + GPR[rs]_{95..64} \times GPR[rt]_{95..64}$
$LO_{63..0} \leftarrow (prod0_{31})^{32} \| prod0_{31..0}$
$HI_{63..0} \leftarrow (prod0_{63})^{32} \| prod0_{63..32}$
$LO_{127..64} \leftarrow (prod1_{31})^{32} \| prod1_{31..0}$
$HI_{127..64} \leftarrow (prod1_{63})^{32} \| prod1_{63..32}$
$GPR[rd]_{63..0} \leftarrow prod0_{63..0}$
$GPR[rd]_{127..64} \leftarrow prod1_{63..0}$



Exceptions:

None

Programming Notes:

See the Programming Notes for the PMADDH instruction.

PMAXH Parallel Maximum Halfword PMAXH

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PMAXH 00111	MMI0 001000
6	5	5	5	5	6

C790

Format: PMAXH rd, rs, rt

Purpose: To select maximum 16-bit signed integers (8 parallel operations).

Description: rd ← max (rs, rt)

The eight signed halfword values in GPR *rt* are subtracted from the corresponding eight signed halfword values in GPR *rs* in parallel. If the result of subtraction is larger than zero, the corresponding signed halfword value in GPR *rs* is placed into the corresponding halfword in GPR *rd* otherwise the corresponding signed halfword value in GPR *rt* is placed into the corresponding halfword of the GPR *rd*.

This instruction operates on 128-bit registers.

Operation:

```
if ((GPR[rs]15..0 – GPR[rt]15..0) > 0) then
  GPR[rd]15..0 ← GPR[rs]15..0
else
  GPR[rd]15..0 ← GPR[rt]15..0
endif

if ((GPR[rs]31..16 – GPR[rt]31..16) > 0) then
  GPR[rd]31..16 ← GPR[rs]31..16
else
  GPR[rd]31..16 ← GPR[rt]31..16
endif

if ((GPR[rs]47..32 – GPR[rt]47..32) > 0) then
  GPR[rd]47..32 ← GPR[rs]47..32
else
  GPR[rd]47..32 ← GPR[rt]47..32
endif

if ((GPR[rs]63..48 – GPR[rt]63..48) > 0) then
  GPR[rd]63..48 ← GPR[rs]63..48
else
  GPR[rd]63..48 ← GPR[rt]63..48
endif

if ((GPR[rs]79..64 – GPR[rt]79..64) > 0) then
  GPR[rd]79..64 ← GPR[rs]79..64
else
  GPR[rd]79..64 ← GPR[rt]79..64
endif
```

if $((GPR[rs]_{95..80} - GPR[rt]_{95..80}) > 0)$ then
  $GPR[rd]_{95..80} \leftarrow GPR[rs]_{95..80}$
else
  $GPR[rd]_{95..80} \leftarrow GPR[rt]_{95..80}$
endif

if $((GPR[rs]_{111..96} - GPR[rt]_{111..96}) > 0)$ then
  $GPR[rd]_{111..96} \leftarrow GPR[rs]_{111..96}$
else
  $GPR[rd]_{111..96} \leftarrow GPR[rt]_{111..96}$
endif

if $((GPR[rs]_{127..112} - GPR[rt]_{127..112}) > 0)$ then
  $GPR[rd]_{127..112} \leftarrow GPR[rs]_{127..112}$
else
  $GPR[rd]_{127..112} \leftarrow GPR[rt]_{127..112}$
endif

	127 112	111 96	95 80	79 64	63 48	47 32	31 16	15 0
rs	A7	A6	A5	A4	A3	A2	A1	A0

	127 112	111 96	95 80	79 64	63 48	47 32	31 16	15 0
rt	B7	B6	B5	B4	B3	B2	B1	B0

	127 112	111 96	95 80	79 64	63 48	47 32	31 16	15 0
rd	max (A7, B7)	max (A6, B6)	max (A5, B5)	max (A4, B4)	max (A3, B3)	max (A2, B2)	max (A1, B1)	max (A0, B0)

Exceptions:

None

PMAXW Parallel Maximum Word PMAXW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PMAXW 00011	MMI0 001000
6	5	5	5	5	6

C790

Format: PMAXW rd, rs, rt

Purpose: To select maximum 32-bit signed integers (4 parallel operations).

Description: rd ← max (rs, rt)

The four signed word values in GPR *rt* are subtracted from the corresponding four signed word values in GPR *rs* in parallel. If the result of subtraction is larger than zero, the corresponding signed word value in GPR *rs* is placed into the corresponding word in GPR *rd* otherwise the corresponding signed word value in GPR *rt* is placed into the corresponding word of the GPR *rd*.

This instruction operates on 128-bit registers.

Operation:

```
if ((GPR[rs]31..0 – GPR[rt]31..0) > 0) then
  GPR[rd]31..0 ← GPR[rs]31..0
else
  GPR[rd]31..0 ← GPR[rt]31..0
endif

if ((GPR[rs]63..32 – GPR[rt]63..32) > 0) then
  GPR[rd]63..32 ← GPR[rs]63..32
else
  GPR[rd]63..32 ← GPR[rt]63..32
endif

if ((GPR[rs]95..64 – GPR[rt]95..64) > 0) then
  GPR[rd]95..64 ← GPR[rs]95..64
else
  GPR[rd]95..64 ← GPR[rt]95..64
endif

if ((GPR[rs]127..96 – GPR[rt]127..96) > 0) then
  GPR[rd]127..96 ← GPR[rs]127..96
else
  GPR[rd]127..96 ← GPR[rt]127..96
endif
```

	127 96	95 64	63 32	31 0
rs	A3	A2	A1	A0

	127 96	95 64	63 32	31 0
rt	B3	B2	B1	B0

	127 96	95 64	63 32	31 0
rd	max (A3, B3)	max (A2, B2)	max (A1, B1)	max (A0, B0)

Exceptions:

None

PMFHI Parallel Move From HI Register PMFHI

31 26	25 16	15 11	10 6	5 0
MMI 011100	0 0000000000	rd	PMFHI 01000	MMI2 001001
6	10	5	5	6

C790

Format: PMFHI rd

Purpose: To copy the special purpose register HI to a GPR.

Description: rd ← HI

The contents of special register *HI* are loaded into GPR *rd.*

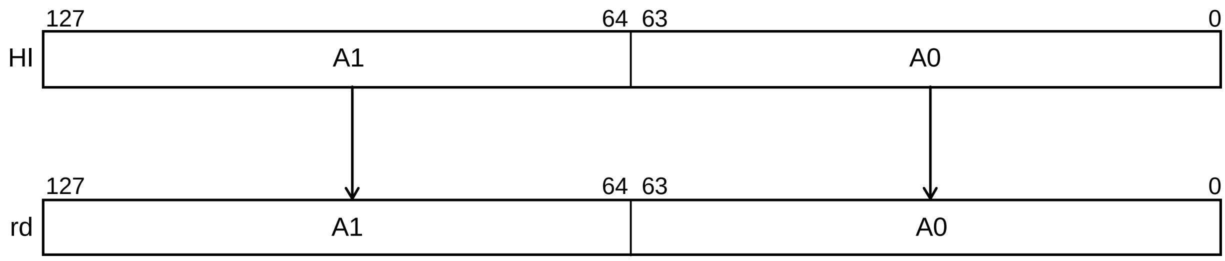
This instruction operates on 128-bit registers.

Restrictions:

None

Operation:

$GPR[rd]_{127..0} \leftarrow HI_{127..0}$



Exceptions:

None

PMFHL.fmt Parallel Move From HI / LO Register PMFHL.fmt

31–26	25–16	15–11	10–6	5–0
MMI 011100	0 0000000000	rd	fmt	PMFHL 110000
6	10	5	5	6

C790

Format: PMFHL.LW rd (fmt = 0)
PMFHL.UW rd (fmt = 1)
PMFHL.SLW rd (fmt = 2)
PMFHL.LH rd (fmt = 3)
PMFHL.SH rd (fmt = 4)

Purpose: To copy the special purpose registers HI / LO to a GPR.

Description: rd ← HI / LO

The contents of special registers *HI* / *LO* are loaded into GPR *rd.*

This instruction operates on 128-bit registers.

Restrictions:

None

Operation:

```
if (fmt = 0) then
  GPR[rd]31..0   ← LO31..0
  GPR[rd]63..32  ← HI31..0
  GPR[rd]95..64  ← LO95..64
  GPR[rd]127..96 ← HI95..64

else if (fmt = 1) then
  GPR[rd]31..0   ← LO63..32
  GPR[rd]63..32  ← HI63..32
  GPR[rd]95..64  ← LO127..96
  GPR[rd]127..96 ← HI127..96

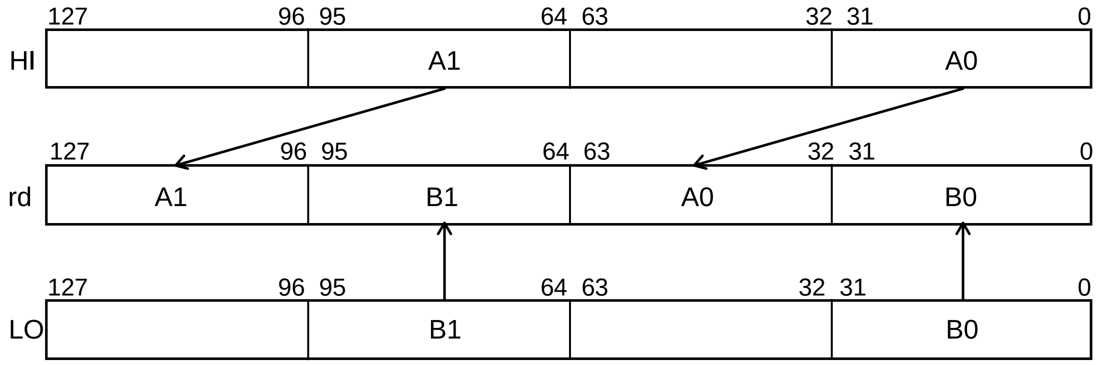
else if (fmt = 2) then
  if (0x7FFFFFFFFFFFFFFF > = (HI31..0 || LO31..0) > 0x000000007FFFFFFF) then
    GPR[rd]63..0← 0x000000007FFFFFFF
  else if (0x8000000000000000 < = (HI31..0 || LO31..0) < -0x0000000080000000) then
    GPR[rd]63..0← 0xFFFFFFFF80000000
  else
    GPR[rd]63..0← HI31..0 || LO31..0
endif
  if ((HI95..64 || LO95..64) > 0x000000007FFFFFFF) then
    GPR[rd]127.. 64← 0x000000007FFFFFFF
  else if ((HI95..64 || LO95..64) < -0x0000000080000000) then
    GPR[rd]127.. 64← -0x0000000080000000
  else
    GPR[rd]127.. 64← (LO95)^32 || LO95..64
endif
else if (fmt = 3) then
  GPR[rd]15..0 ← LO15..0
```

```
    GPR[rd]_{31..16} ← LO_{47..32}
    GPR[rd]_{47..32} ← HI_{15..0}
    GPR[rd]_{63..48} ← HI_{47..32}
    GPR[rd]_{79..64} ← LO_{79..64}
    GPR[rd]_{95..80} ← LO_{111..96}
    GPR[rd]_{111..96} ← HI_{79..64}
    GPR[rd]_{127..112} ← HI_{111..96}

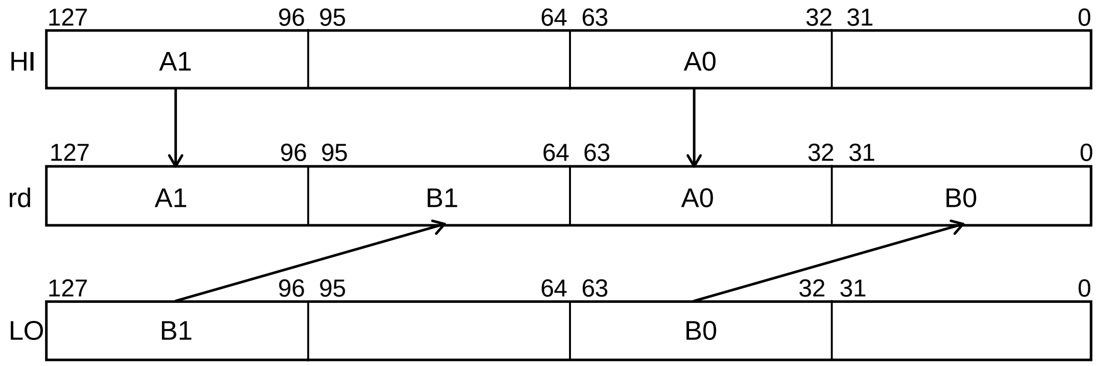
  else if (fmt = 4) then
    if (0x7FFFFFF> = LO_{31..0} > 0x00007FFF) then
      GPR[rd]_{15..0} ← 0x7FFF
    else if (0x80000000< = LO_{31..0} < 0xFFFF8000) then
      GPR[rd]_{15..0} ← 0x8000
    else
      GPR[rd]_{15..0} ← LO_{15..0}
    endif
    if (LO_{63..32} > 0x00007FFF) then
      GPR[rd]_{31..16} ← 0x7FFF
    else if (LO_{63..32} < 0xFFFF8000) then
      GPR[rd]_{31..16} ← 0x8000
    else
      GPR[rd]_{31..16} ← LO_{47..32}
    endif
    if (HI_{31..0} > 0x00007FFF) then
      GPR[rd]_{47..32} ← 0x7FFF
    else if (HI_{31..0} < 0xFFFF8000) then
      GPR[rd]_{47..32} ← 0x8000
    else
      GPR[rd]_{47..32} ← HI_{15..0}
    endif
    if (HI_{63..32} > 0x00007FFF) then
      GPR[rd]_{63..48} ← 0x7FFF
    else if (HI_{63..32} < 0xFFFF8000) then
      GPR[rd]_{63..48} ← 0x8000
    else
      GPR[rd]_{63..48} ← HI_{47..32}
    endif
    if (LO_{95..64} > 0x00007FFF) then
      GPR[rd]_{79..64} ← 0x7FFF
    else if (LO_{95..64} < -0xFFFF8000) then
      GPR[rd]_{79..64} ← 0x8000
    else
      GPR[rd]_{79..64} ← LO_{79..64}
    endif
    if (LO_{127..96} > 0x00007FFF) then
      GPR[rd]_{95..80} ← 0x7FFF
    else if (LO_{127..96} < 0xFFFF8000) then
      GPR[rd]_{95..80} ← 0x8000
    else
      GPR[rd]_{95..80} ← LO_{111..96}
    endif
    if (HI_{95..64} > 0x00007FFF) then
      GPR[rd]_{111..96} ← 0x7FFF
    else if (HI_{95..64} < 0xFFFF8000) then
      GPR[rd]_{111..96} ← 0x8000
```

```
    else
      GPR[rd]111..96← HI79..64
    endif
    if (HI127..96 > 0x00007FFF) then
      GPR[rd]127..112← 0x7FFF
    else if (HI127..96 < 0xFFFF8000) then
      GPR[rd]127..112← 0x8000
    else
      GPR[rd]127..112← HI111..96
    endif
  endif
```

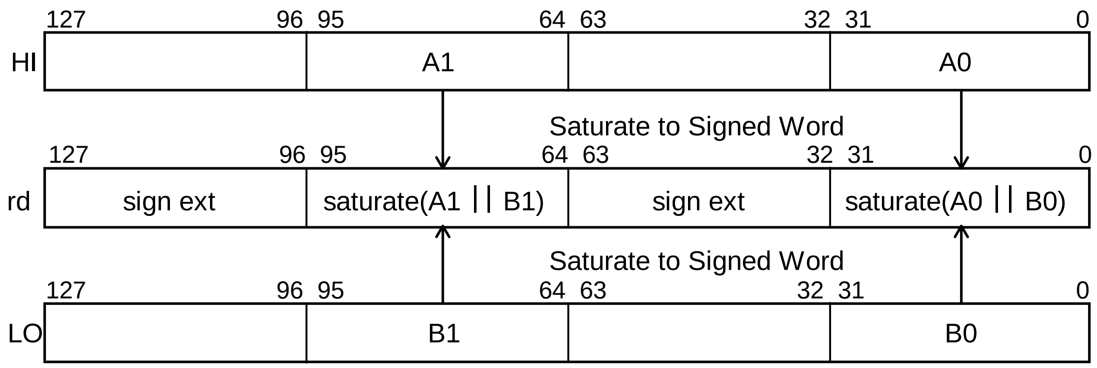
(fmt = 0)



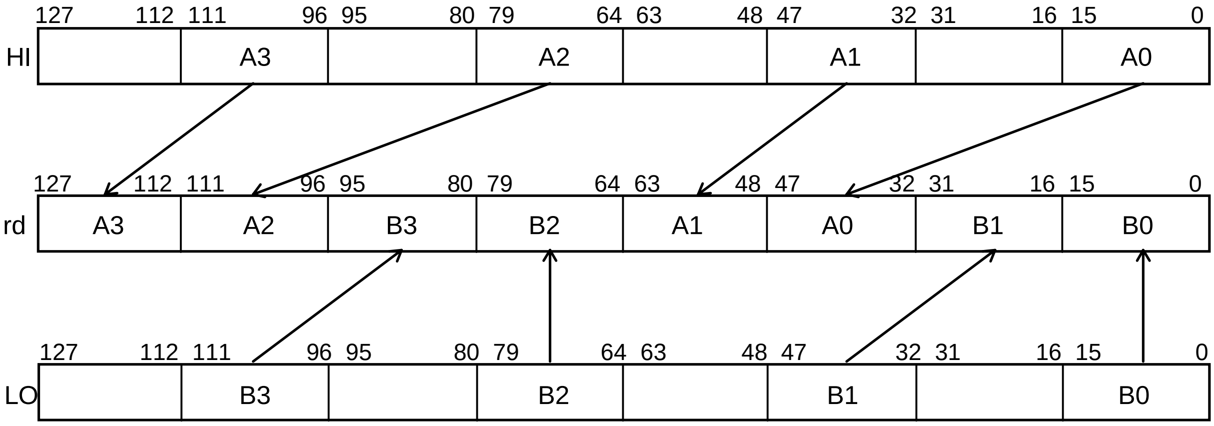
(fmt = 1)



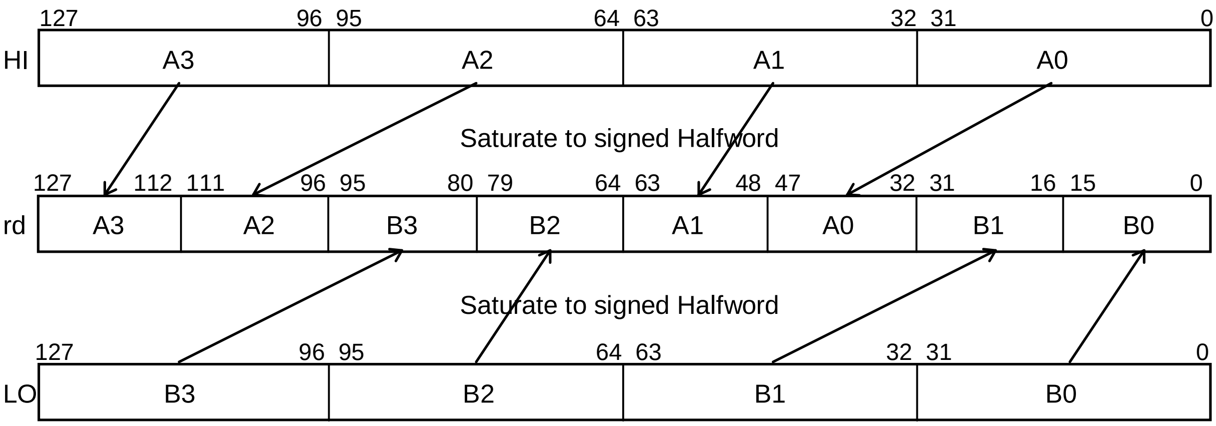
(fmt = 2)



(fmt = 3)



(fmt = 4)



Exceptions:

None

PMFLO Parallel Move From LO Register PMFLO

31 26	25 16	15 11	10 6	5 0
MMI 011100	0 0000000000	rd	PMFLO 01001	MMI2 001001
6	10	5	5	6

C790

Format: PMFLO rd

Purpose: To copy the special purpose register LO to a GPR.

Description: rd ← LO

The contents of special register *LO* are loaded into GPR *rd*.

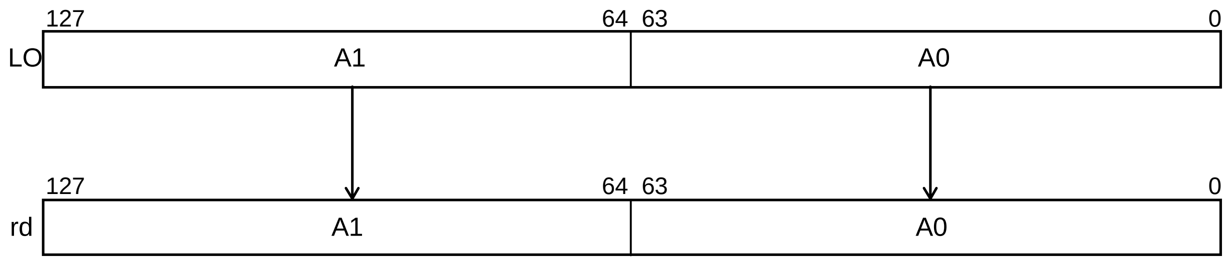
This instruction operates on 128-bit registers.

Restrictions:

None

Operation:

$GPR[rd]_{127..0} \leftarrow LO_{127..0}$



Exceptions:

None

PMINH — Parallel Minimum Halfword — PMINH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PMINH 00111	MMI1 101000
6	5	5	5	5	6

C790

Format: PMINH rd, rs, rt

Purpose: To select the minimum of two 16-bit signed integers (8 parallel operations).

Description: rd ← min (rs, rt)

The eight signed halfword values in GPR *rt* are subtracted from the corresponding eight signed halfword values in GPR *rs* in parallel. If the result of each subtraction is larger than zero, the corresponding signed halfword in GPR *rt* is placed into the corresponding halfword in GPR *rd* otherwise the corresponding signed halfword in GPR *rs* is placed into the corresponding halfword of GPR *rd.*

This instruction operates on 128-bit registers.

Operation:

```
if ((GPR[rs]15..0 – GPR[rt]15..0) > 0) then
  GPR[rd]15..0 ← GPR[rt]15..0
else
  GPR[rd]15..0 ← GPR[rs]15..0
endif
if ((GPR[rs]31..16 – GPR[rt]31..16) > 0) then
  GPR[rd]31..16 ← GPR[rt]31..16
else
  GPR[rd]31..16 ← GPR[rs]31..16
endif
if ((GPR[rs]47..32 – GPR[rt]47..32) > 0) then
  GPR[rd]47..32 ← GPR[rt]47..32
else
  GPR[rd]47..32 ← GPR[rs]47..32
endif
if ((GPR[rs]63..48 – GPR[rt]63..48) > 0) then
  GPR[rd]63..48 ← GPR[rt]63..48
else
  GPR[rd]63..48 ← GPR[rs]63..48
endif
if ((GPR[rs]79..64 – GPR[rt]79..64) > 0) then
  GPR[rd]79..64 ← GPR[rt]79..64
else
  GPR[rd]79..64 ← GPR[rs]79..64
endif
if ((GPR[rs]95..80 – GPR[rt]95..80) > 0) then
  GPR[rd]95..80 ← GPR[rt]95..80
else
  GPR[rd]95..80 ← GPR[rs]95..80
endif
```

if (($GPR[rs]_{111..96} - GPR[rt]_{111..96}$) > 0) then
  $GPR[rd]_{111..96} \leftarrow GPR[rt]_{111..96}$
else
  $GPR[rd]_{111..96} \leftarrow GPR[rs]_{111..96}$
endif
if (($GPR[rs]_{127..112} - GPR[rt]_{127..112}$) > 0) then
  $GPR[rd]_{127..112} \leftarrow GPR[rt]_{127..112}$
else
  $GPR[rd]_{127..112} \leftarrow GPR[rs]_{127..112}$
endif

	127 112	111 96	95 80	79 64	63 48	47 32	31 16	15 0
rs	A7	A6	A5	A4	A3	A2	A1	A0

	127 112	111 96	95 80	79 64	63 48	47 32	31 16	15 0
rt	B7	B6	B5	B4	B3	B2	B1	B0

	127 112	111 96	95 80	79 64	63 48	47 32	31 16	15 0
rd	min (A7, B7)	min (A6, B6)	min (A5, B5)	min (A4, B4)	min (A3, B3)	min (A2, B2)	min (A1, B1)	min (A0, B0)

Exceptions:

None

PMINW Parallel Minimum Word PMINW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PMINW 00011	MMI1 101000
6	5	5	5	5	6

C790

Format: PMINW rd, rs, rt

Purpose: To select the minimum of two 32-bit signed integers (4 parallel operations).

Description: rd ← min (rs, rt)

The four signed word values in GPR *rt* are subtracts from the corresponding four signed word values in GPR *rs,* in parallel. If the result of each subtraction is larger than zero, the corresponding signed word value in GPR *rt* is placed into the corresponding word of GPR *rd* otherwise the corresponding signed word value in GPR *rs* is placed into the corresponding word of GPR *rd.*

This instruction operates on 128-bit registers.

Operation:

```
if ((GPR[rs]31..0 – GPR[rt]31..0) > 0) then
  GPR[rd]31..0 ← GPR[rt]31..0
else
  GPR[rd]31..0 ← GPR[rs]31..0
endif

if ((GPR[rs]63..32 – GPR[rt]63..32) > 0) then
  GPR[rd]63..32 ← GPR[rt]63..32
else
  GPR[rd]63..32 ← GPR[rs]63..32
endif

if ((GPR[rs]95..64 – GPR[rt]95..64) > 0) then
  GPR[rd]95..64 ← GPR[rt]95..64
else
  GPR[rd]95..64 ← GPR[rs]95..64
endif

if ((GPR[rs]127..96 – GPR[rt]127..96) > 0) then
  GPR[rd]127..96 ← GPR[rt]127..96
else
  GPR[rd]127..96 ← GPR[rs]127..96
endif
```

	127 ... 96	95 ... 64	63 ... 32	31 ... 0
rs	A3	A2	A1	A0
rt	B3	B2	B1	B0

	127 ... 96	95 ... 64	63 ... 32	31 ... 0
rd	min (A3, B3)	min (A2, B2)	min (A1, B1)	min (A0, B0)

Exceptions:

None

PMSUBH Parallel Multiply-Subtract Halfword PMSUBH

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PMSUBH 10100	MMI2 001001
6	5	5	5	5	6

C790

Format: PMSUBH rd, rs, rt

Purpose: To multiply 8 pairs of 16-bit signed integers and subtract in parallel.

Description: (rd, HI, LO) ← (HI, LO) – rs × rt

The eight signed halfwords in GPR *rs* are multiplied by the eight signed halfwords in GPR *rt* in parallel. The eight word multiply results are subtracted from the corresponding words in special registers *HI* and *LO*, and the word results are placed into the corresponding words in special registers *HI*, *LO* and GPR *rd*.

No arithmetic exception occurs under any circumstances.

This instruction operates on 128-bit registers.

Restrictions:

None

Operation:

prod0 ← $LO_{31..0}$ – $GPR[rs]_{15..0}$ × $GPR[rt]_{15..0}$
prod1 ← $LO_{63..32}$ – $GPR[rs]_{31..16}$ × $GPR[rt]_{31..16}$
prod2 ← $HI_{31..0}$ – $GPR[rs]_{47..32}$ × $GPR[rt]_{47..32}$
prod3 ← $HI_{63..32}$ – $GPR[rs]_{63..48}$ × $GPR[rt]_{63..48}$
prod4 ← $LO_{95..64}$ – $GPR[rs]_{79..64}$ × $GPR[rt]_{79..64}$
prod5 ← $LO_{127..96}$ – $GPR[rs]_{95..80}$ × $GPR[rt]_{95..80}$
prod6 ← $HI_{95..64}$ – $GPR[rs]_{111..96}$ × $GPR[rt]_{111..96}$
prod7 ← $HI_{127..96}$ – $GPR[rs]_{127..112}$ × $GPR[rt]_{127..112}$
$LO_{31..0}$ ← $prod0_{31..0}$
$LO_{63..32}$ ← $prod1_{31..0}$
$HI_{31..0}$ ← $prod2_{31..0}$
$HI_{63..32}$ ← $prod3_{31..0}$
$LO_{95..64}$ ← $prod4_{31..0}$
$LO_{127..96}$ ← $prod5_{31..0}$
$HI_{95..64}$ ← $prod6_{31..0}$
$HI_{127..96}$ ← $prod7_{31..0}$
$GPR[rd]_{31..0}$ ← $prod0_{31..0}$
$GPR[rd]_{63..32}$ ← $prod2_{31..0}$
$GPR[rd]_{95..64}$ ← $prod4_{31..0}$
$GPR[rd]_{127..96}$ ← $prod6_{31..0}$

	127 112	111 96	95 80	79 64	63 48	47 32	31 16	15 0
rs	A7	A6	A5	A4	A3	A2	A1	A0
	×	×	×	×	×	×	×	×
rt	B7	B6	B5	B4	B3	B2	B1	B0

	127 96	95 64	63 32	31 0
HI	C7	C6	C3	C2
LO	C5	C4	C1	C0

---

	127 96	95 64	63 32	31 0
rd	C6 – A6 × B6	C4 – A4 × B4	C2 – A2 × B2	C0 – A0 × B0

	127 96	95 64	63 32	31 0
HI	C7 – A7 × B7	C6 – A6 × B6	C3 – A3 × B3	C2 – A2 × B2
LO	C5 – A5 × B5	C4 – A4 × B4	C1 – A1 × B1	C0 – A0 × B0

Exceptions:

None

Programming Notes:

See the Programming Notes for the PMADDH instruction.

PMSUBW   Parallel Multiply-Subtract Word   PMSUBW

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PMSUBW 00100	MMI2 001001
6	5	5	5	5	6

C790

Format: PMSUBW rd, rs, rt

Purpose: To multiply 2 pairs of 32-bit signed integers and subtract in parallel.

Description: (rd, HI, LO) ← (HI, LO) – rs × rt

The low-order signed words of the two doublewords in GPR *rs* are multiplied by the low-order signed words of the two doublewords in GPR *rt* in parallel. The two 64-bit multiply results are subtracted from the contents of special registers *HI* and *LO*. The low-order word of the two doubleword results are placed into special register *LO*, and the high-order word of the two doubleword results are placed into special register *HI*. The two doubleword results are placed into GPR *rd*.

No arithmetic exception occurs under any circumstances.

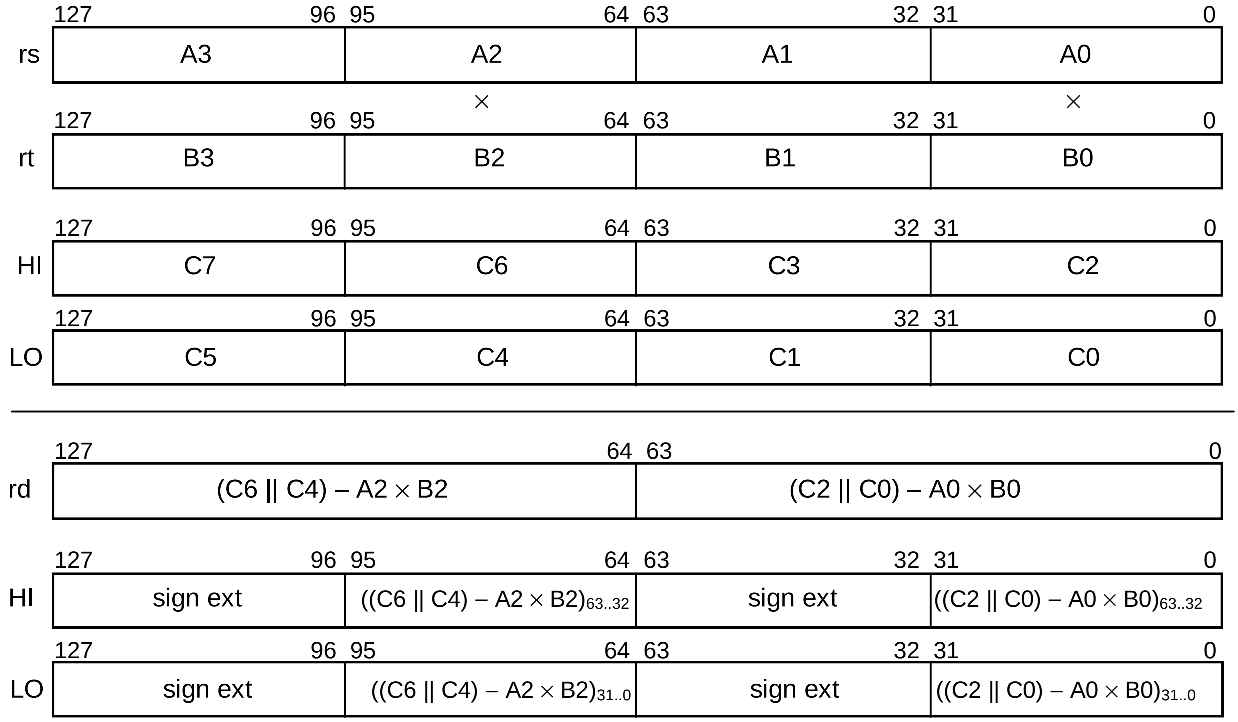
This instruction operates on 128-bit registers.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 127..95 and 63..31 equal) then the result of the equation will be undefined.

Operation:

if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
prod0 ← $(HI_{31..0} \| LO_{31..0}) - GPR[rs]_{31..0} \times GPR[rt]_{31..0}$
prod1 ← $(HI_{95..64} \| LO_{95..64}) - GPR[rs]_{95..64} \times GPR[rt]_{95..64}$
$LO_{63..0}$ ← $(prod0_{31})^{32} \| prod0_{31..0}$
$HI_{63..0}$ ← $(prod0_{63})^{32} \| prod0_{63..32}$
$LO_{127..64}$ ← $(prod1_{31})^{32} \| prod1_{31..0}$
$HI_{127..64}$ ← $(prod1_{63})^{32} \| prod1_{63..32}$
$GPR[rd]_{63..0}$ ← $prod0_{63..0}$
$GPR[rd]_{127..64}$ ← $prod1_{63..0}$



Exceptions:

None

Programming Notes:

See the Programming Notes for the PMADDH instruction.

PMTHI

Parallel Move To HI Register

PMTHI

31 26	25 21	20 11	10 6	5 0
MMI 011100	rs	0 0000000000	PMTHI 01000	MMI3 101001
6	5	10	5	6

C790

Format: PMTHI rs

Purpose: To copy a GPR to the special purpose register HI.

Description: HI ← rs

The contents of GPR *rs* are loaded into special register *HI*.

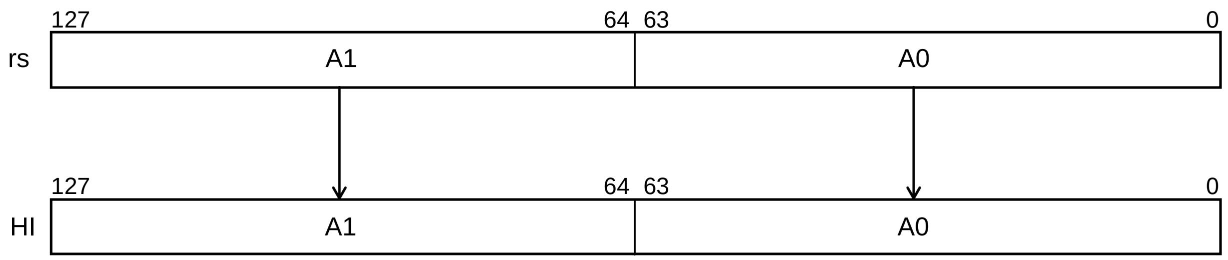
This instruction operates on 128-bit registers.

Restrictions:

None

Operation:

$HI_{127..0}$ $\leftarrow GPR[rs]_{127..0}$



Exceptions:

None

PMTHL.fmt Parallel Move To HI / LO Register PMTHL.fmt

31–26	25–21	20–11	10–6	5–0
MMI 011100	rs	0 0000000000	fmt	PMTHL 110001
6	5	10	5	6

C790

Format: PMTHL.LW rs (fmt = 0)

Purpose: To copy a GPR to the special registers HI / LO.

Description: HI / LO ← rs

The contents of GPR *rd* are loaded into special register *HI* / *LO*.

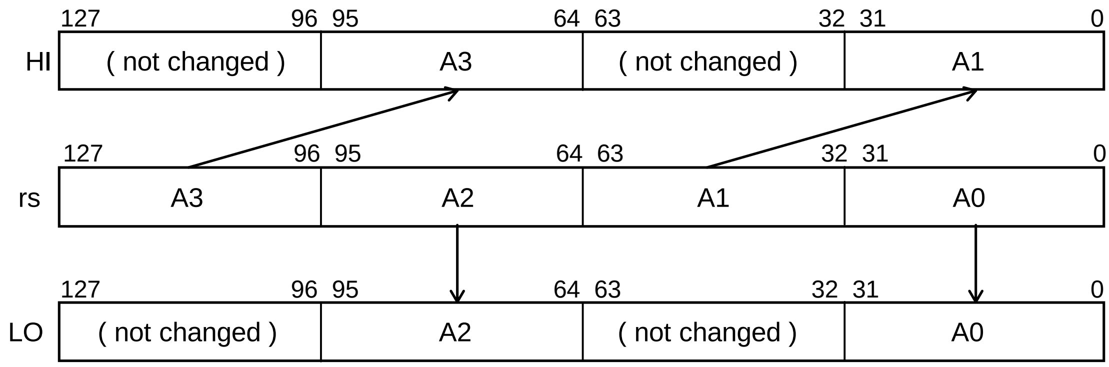
This instruction operates on 128-bit registers.

Restrictions:

None

Operation:

if (fmt = 0) then
$LO_{31..0} \leftarrow GPR[rs]_{31..0}$
$LO_{63..32} \leftarrow LO_{63..32}$
$HI_{31..0} \leftarrow GPR[rs]_{63..32}$
$HI_{63..32} \leftarrow HI_{63..32}$
$LO_{95..64} \leftarrow GPR[rs]_{95..64}$
$LO_{127..96} \leftarrow LO_{127..96}$
$HI_{95..64} \leftarrow GPR[rs]_{127..96}$
$HI_{127..96} \leftarrow HI_{127..96}$
endif



Exceptions:

None

PMTLO Parallel Move To LO Register PMTLO

31–26	25–21	20–11	10–6	5–0
MMI 011100	rs	0 0000000000	PMTLO 01001	MMI3 101001
6	5	10	5	6

C790

Format: PMTLO rs

Purpose: To copy a GPR to the special register LO.

Description: LO ← rs

The contents of GPR *rs* are loaded into special register *LO*.

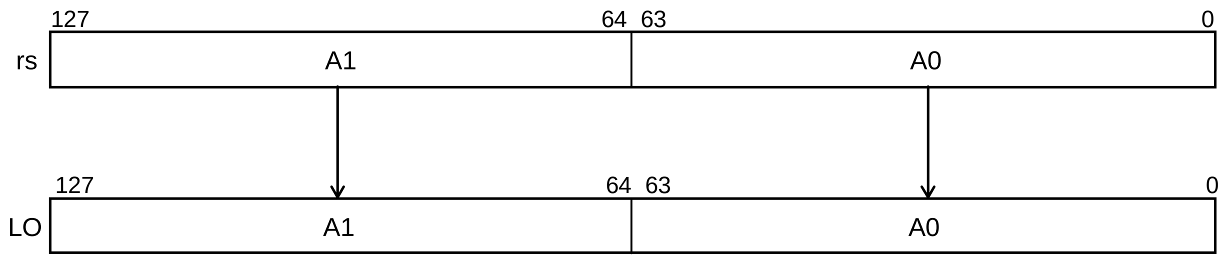
This instruction operates on 128-bit registers.

Restrictions:

None

Operation:

$LO_{127..0} \leftarrow GPR[rs]_{127..0}$



Exceptions:

None

PMULTH Parallel Multiply Halfword PMULTH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PMULTH 11100	MMI2 001001
6	5	5	5	5	6

C790

Format: PMULTH rd, rs, rt

Purpose: To multiply 8 pairs of 16-bit signed integers in parallel.

Description: (rd, LO, HI) ← rs × rt

The eight signed halfwords in GPR *rs* are multiplied by the eight signed halfwords in GPR *rt,* in parallel. The eight word results are placed into special register *HI*, *LO* and GPR *rd.*

No arithmetic exception occurs under any circumstances.

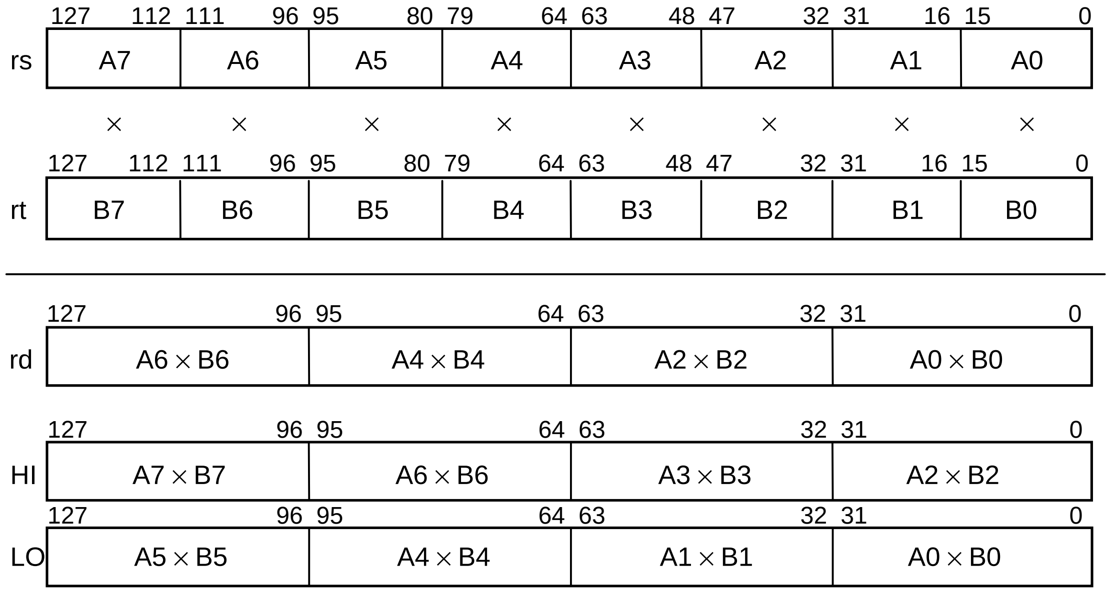
This instruction operates on 128-bit registers.

Restrictions:

None

Operation:

$prod0 \leftarrow GPR[rs]_{15..0} \times GPR[rt]_{15..0}$
$prod1 \leftarrow GPR[rs]_{31..16} \times GPR[rt]_{31..16}$
$prod2 \leftarrow GPR[rs]_{47..32} \times GPR[rt]_{47..32}$
$prod3 \leftarrow GPR[rs]_{63..48} \times GPR[rt]_{63..48}$
$prod4 \leftarrow GPR[rs]_{79..64} \times GPR[rt]_{79..64}$
$prod5 \leftarrow GPR[rs]_{95..80} \times GPR[rt]_{95..80}$
$prod6 \leftarrow GPR[rs]_{111..96} \times GPR[rt]_{111..96}$
$prod7 \leftarrow GPR[rs]_{127..112} \times GPR[rt]_{127..112}$
$LO_{31..0} \leftarrow prod0_{31..0}$
$LO_{63..32} \leftarrow prod1_{31..0}$
$HI_{31..0} \leftarrow prod2_{31..0}$
$HI_{63..32} \leftarrow prod3_{31..0}$
$LO_{95..64} \leftarrow prod4_{31..0}$
$LO_{127..96} \leftarrow prod5_{31..0}$
$HI_{95..64} \leftarrow prod6_{31..0}$
$HI_{127..96} \leftarrow prod7_{31..0}$
$GPR[rd]_{31..0} \leftarrow prod0_{31..0}$
$GPR[rd]_{63..32} \leftarrow prod2_{31..0}$
$GPR[rd]_{95..64} \leftarrow prod4_{31..0}$
$GPR[rd]_{127..96} \leftarrow prod6_{31..0}$



Exceptions:

None

Programming Notes:

See the Programming Notes of the PMADDH instruction.

PMULTUW   Parallel Multiply Unsigned Word   PMULTUW

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PMULTUW 01100	MMI3 101001
6	5	5	5	5	6

C790

Format: PMULTUW rd, rs, rt

Purpose: To multiply 2 pairs of 32-bit unsigned integers in parallel.

Description: (rd, LO, HI) $\leftarrow$ rs $\times$ rt

The low-order unsigned words of the two doublewords in GPR *rs* are multiplied by the low-order unsigned words of the two doublewords in GPR *rt* in parallel. The low-order word of the two doubleword result is placed into special register *LO*, and the high-order word of the two doubleword result is placed into special register *HI*. The two doubleword results are placed into GPR *rd*.

No arithmetic exception occurs under any circumstances.

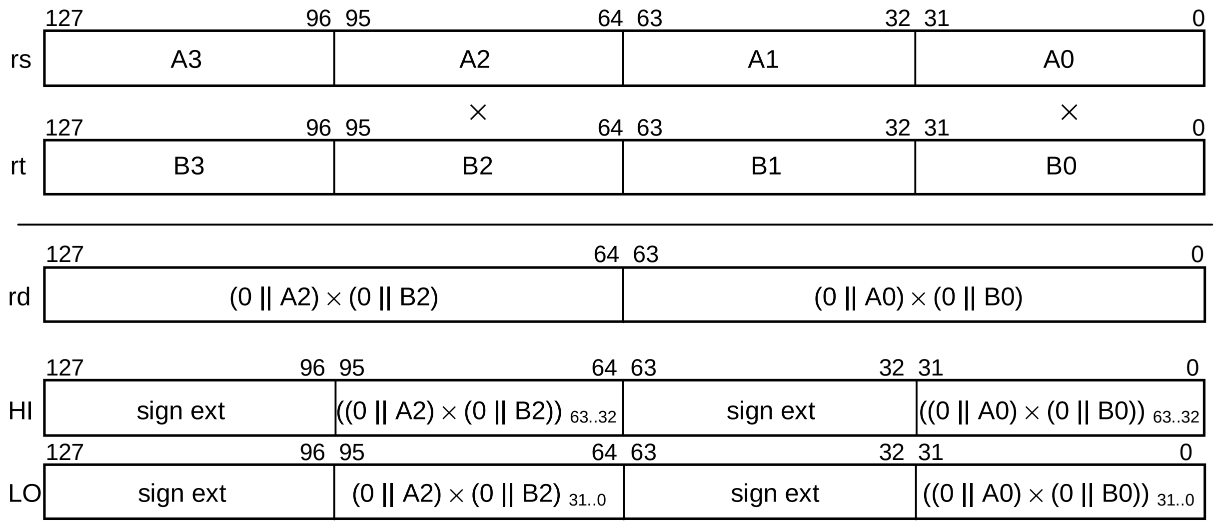
This instruction operates on 128-bit registers.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain zero-extended 32-bit values (bits 127..96 and 63..32 equal zero) then the result of the equation will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod0 $\leftarrow$ $(0 \,\|\, GPR[rs]_{31..0}) \times (0 \,\|\, GPR[rt]_{31..0})$
prod1 $\leftarrow$ $(0 \,\|\, GPR[rs]_{95..64}) \times (0 \,\|\, GPR[rt]_{95..64})$
$LO_{63..0}$ $\leftarrow$ $(prod0_{31})^{32} \,\|\, prod0_{31..0}$
$HI_{63..0}$ $\leftarrow$ $(prod0_{63})^{32} \,\|\, prod0_{63..32}$
$LO_{127..64}$ $\leftarrow$ $(prod1_{31})^{32} \,\|\, prod1_{31..0}$
$HI_{127..64}$ $\leftarrow$ $(prod1_{63})^{32} \,\|\, prod1_{63..32}$
$GPR[rd]_{63..0}$ $\leftarrow$ prod0
$GPR[rd]_{127..64}$ $\leftarrow$ prod1



Exceptions:

None

Programming Notes:

See the Programming Notes of the PMADDH instruction.

PMULTW — Parallel Multiply Word — PMULTW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PMULTW 01100	MMI2 001001
6	5	5	5	5	6

C790

Format: PMULTW rd, rs, rt

Purpose: To multiply 2 pairs of 32-bit signed integers in parallel.

Description: (rd, LO, HI) ← rs × rt

The low-order signed words of the two doublewords in GPR *rs* are multiplied by the low-order signed words of the two doublewords in GPR *rt* in parallel. The low-order word of the two doubleword results is placed into special register *LO*, and the high-order word of the two doubleword results is placed into special register *HI*. The two doubleword results are placed into GPR *rd*.

No arithmetic exception occurs under any circumstances.

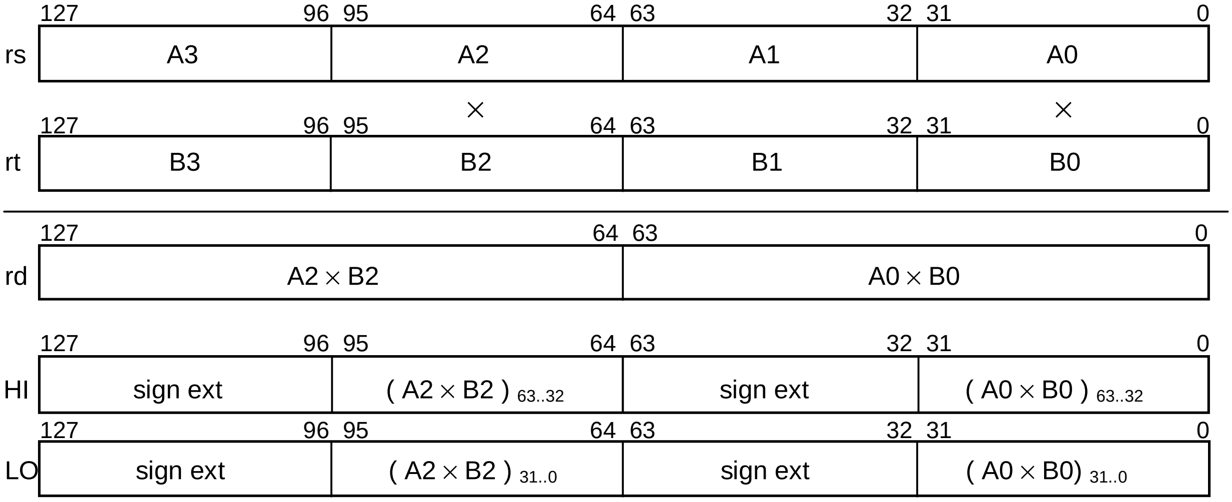
This instruction operates on 128-bit registers.

Restrictions:

If either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 127..95 and 63..31 equal) then the result of the equation will be undefined.

Operation:

if (NotWordValue (GPR[rs]) or NotWordValue (GPR[rt])) then UndefinedResult() endif
prod0 ← $GPR[rs]_{31..0} \times GPR[rt]_{31..0}$
prod1 ← $GPR[rs]_{95..64} \times GPR[rt]_{95..64}$
$LO_{63..0}$ ← $(prod0_{31})^{32} \| prod0_{31..0}$
$HI_{63..0}$ ← $(prod0_{63})^{32} \| prod0_{63..32}$
$LO_{127..64}$ ← $(prod1_{31})^{32} \| prod1_{31..0}$
$HI_{127..64}$ ← $(prod1_{63})^{32} \| prod1_{63..32}$
$GPR[rd]_{63..0}$ ← prod0
$GPR[rd]_{127..64}$ ← prod1



Exceptions:

None

Programming Notes:

See the Programming Notes of the PMADDH instruction.

PNOR

Parallel Not Or

PNOR

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PNOR 10011	MMI3 101001
6	5	5	5	5	6

C790

Format: PNOR rd, rs, rt

Purpose: To do a bitwise logical NOT OR (NOR).

Description: rd ← rs NOR rt

The contents of GPR *rs* are combined with the contents of GPR rt in a bitwise logical NOR operation. The result is placed into GPR *rd*.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{127..0} \leftarrow GPR[rs]_{127..0}$ nor $GPR[rt]_{127..0}$

	127–64	63–0
rs	A1	A0
	NOR	NOR
rt	B1	B0
rd	A1 NOR B1	A0 NOR B0

Exceptions:

None

POR — Parallel Or — POR

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	POR 10010	MMI3 101001
6	5	5	5	5	6

C790

Format: POR rd, rs, rt

Purpose: To do a bitwise logical OR.

Description: rd ← rs OR rt

The contents of GPR *rs* are combined with the contents of GPR rt in a bitwise logical OR operation. The result is placed into GPR *rd.*

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{127..0} \leftarrow GPR[rs]_{127..0}$ or $GPR[rt]_{127..0}$

	127 … 64	63 … 0
rs	A1	A0
	OR	OR
rt	B1	B0
rd	A1 OR B1	A0 OR B0

Exceptions:

None

PPAC5 Parallel Pack to 5-bits PPAC5

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	0 00000	rt	rd	PPAC5 11111	MMI0 001000
6	5	5	5	5	6

C790

Format: PPAC5 rd, rt

Purpose: To truncate and pack data into consecutive 5-bits.

Description: rd ← pack (rt)

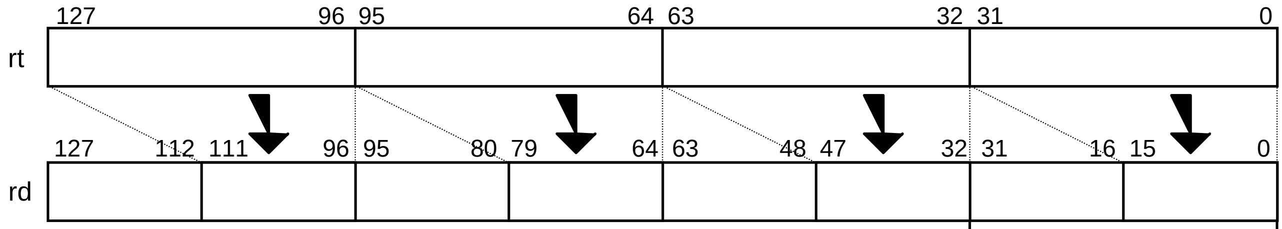
The four 32-bit words (8, 8, 8, 8 bit) in GPR *rt* are packed into the four 16-bit halfwords (1, 5, 5, 5 bit). The results are placed into GPR *rd*. See diagram on next page.

This instruction operates on 128-bit registers.

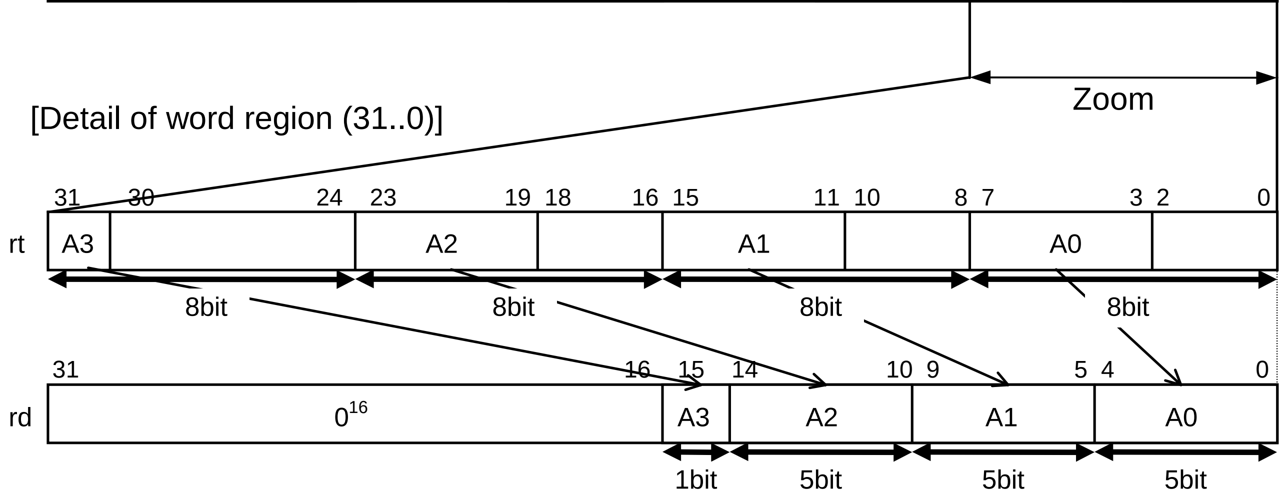
Operation

$GPR[rd]_{4..0} \leftarrow GPR[rt]_{7..3}$
$GPR[rd]_{9..5} \leftarrow GPR[rt]_{15..11}$
$GPR[rd]_{14..10} \leftarrow GPR[rt]_{23..19}$
$GPR[rd]_{15} \leftarrow GPR[rt]_{31}$
$GPR[rd]_{31..16} \leftarrow 0^{16}$
$GPR[rd]_{36..32} \leftarrow GPR[rt]_{39..35}$
$GPR[rd]_{41..37} \leftarrow GPR[rt]_{47..43}$
$GPR[rd]_{46..42} \leftarrow GPR[rt]_{55..51}$
$GPR[rd]_{47} \leftarrow GPR[rt]_{63}$
$GPR[rd]_{63..48} \leftarrow 0^{16}$
$GPR[rd]_{68..64} \leftarrow GPR[rt]_{71..67}$
$GPR[rd]_{73..69} \leftarrow GPR[rt]_{79..75}$
$GPR[rd]_{78..74} \leftarrow GPR[rt]_{87..83}$
$GPR[rd]_{79} \leftarrow GPR[rt]_{95}$
$GPR[rd]_{95..80} \leftarrow 0^{16}$
$GPR[rd]_{100..96} \leftarrow GPR[rt]_{103..99}$
$GPR[rd]_{105..101} \leftarrow GPR[rt]_{111..107}$
$GPR[rd]_{110..106} \leftarrow GPR[rt]_{119..115}$
$GPR[rd]_{111} \leftarrow GPR[rt]_{127}$
$GPR[rd]_{127..112} \leftarrow 0^{16}$

[Overview]



[Detail of word region (31..0)]



Exceptions:

None

PPACB — Parallel Pack to Byte — PPACB

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PPACB 11011	MMI0 001000
6	5	5	5	5	6

C790

Format: PPACB rd, rs, rt

Purpose: To pack into consecutive bytes.

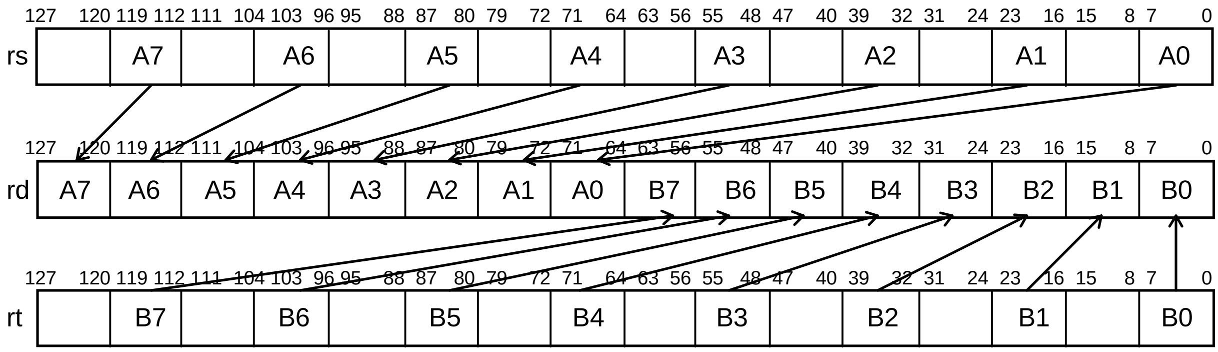
Description: rd ← pack (rs, rt)

The low-order bytes of the eight halfwords in GPR *rs* are packed into consecutive bytes of the high-order doubleword in GPR rd. Similarly, the low-order bytes of the eight halfwords in GPR *rt* are packed into consecutive bytes of the low-order doubleword in GPR rd.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{7..0} \leftarrow GPR[rt]_{7..0}$
$GPR[rd]_{15..8} \leftarrow GPR[rt]_{23..16}$
$GPR[rd]_{23..16} \leftarrow GPR[rt]_{39..32}$
$GPR[rd]_{31..24} \leftarrow GPR[rt]_{55..48}$
$GPR[rd]_{39..32} \leftarrow GPR[rt]_{71..64}$
$GPR[rd]_{47..40} \leftarrow GPR[rt]_{87..80}$
$GPR[rd]_{55..48} \leftarrow GPR[rt]_{103..96}$
$GPR[rd]_{63..56} \leftarrow GPR[rt]_{119..112}$
$GPR[rd]_{71..64} \leftarrow GPR[rs]_{7..0}$
$GPR[rd]_{79..72} \leftarrow GPR[rs]_{23..16}$
$GPR[rd]_{87..80} \leftarrow GPR[rs]_{39..32}$
$GPR[rd]_{95..88} \leftarrow GPR[rs]_{55..48}$
$GPR[rd]_{103..96} \leftarrow GPR[rs]_{71..64}$
$GPR[rd]_{111..104} \leftarrow GPR[rs]_{87..80}$
$GPR[rd]_{119..112} \leftarrow GPR[rs]_{103..96}$
$GPR[rd]_{127..120} \leftarrow GPR[rs]_{119..112}$



Exceptions:

None

PPACH Parallel Pack to Halfword PPACH

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PPACH 10111	MMI0 001000
6	5	5	5	5	6

C790

Format: PPACH rd, rs, rt

Purpose: To pack into consecutive halfwords.

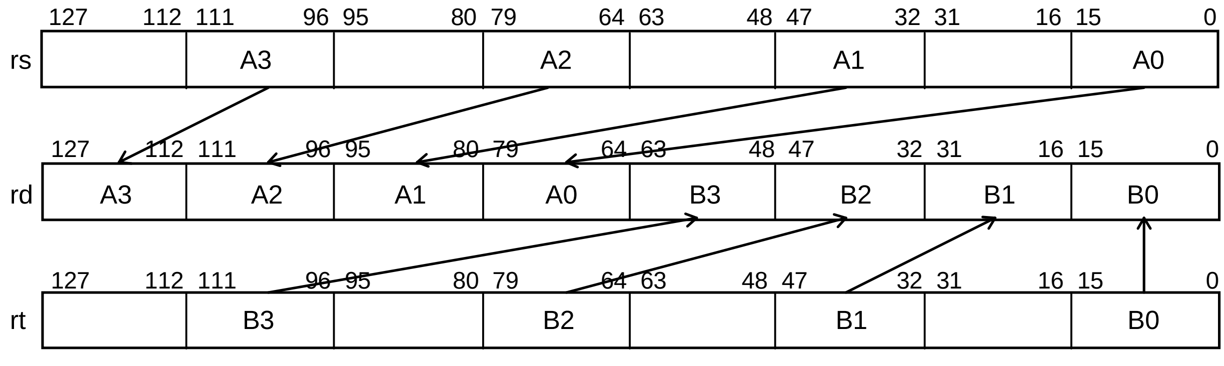
Description: rd ← pack (rs, rt)

The low-order halfwords of the four words in GPR *rs* are packed into consecutive halfwords of the high-order doubleword in GPR rd. Similarly, the low-order halfwords of the four words in GPR *rt* are packed into consecutive halfwords of the low-order doubleword in GPR rd.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{15..0} \leftarrow GPR[rt]_{15..0}$
$GPR[rd]_{31..16} \leftarrow GPR[rt]_{47..32}$
$GPR[rd]_{47..32} \leftarrow GPR[rt]_{79..64}$
$GPR[rd]_{63..48} \leftarrow GPR[rt]_{111..96}$
$GPR[rd]_{79..64} \leftarrow GPR[rs]_{15..0}$
$GPR[rd]_{95..80} \leftarrow GPR[rs]_{47..32}$
$GPR[rd]_{111..96} \leftarrow GPR[rs]_{79..64}$
$GPR[rd]_{127..112} \leftarrow GPR[rs]_{111..96}$



Exceptions:

None

PPACW

Parallel Pack to Word

PPACW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PPACW 10011	MMI0 001000
6	5	5	5	5	6

C790

Format: PPACW rd, rs, rt

Purpose: To pack into consecutive words.

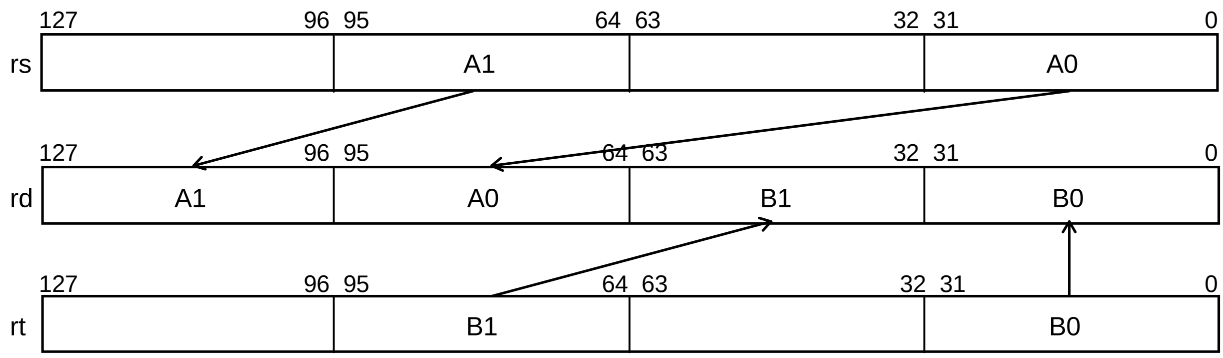
Description: rd ← pack (rs, rt)

The low-order words of the two doublewords in GPR *rs* are packed into consecutive words of the high-order doubleword in GPR rd. Similarly, the low-order words of the two doublewords in GPR *rt* are packed into consecutive words of the low-order doubleword in GPR rd.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{31..0} \leftarrow GPR[rt]_{31..0}$
$GPR[rd]_{63..32} \leftarrow GPR[rt]_{95..64}$
$GPR[rd]_{95..64} \leftarrow GPR[rs]_{31..0}$
$GPR[rd]_{127..96} \leftarrow GPR[rs]_{95..64}$



Exceptions:

None

PREVH Parallel Reverse Halfword PREVH

31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0
MMI 011100	0 00000	rt	rd	PREVH 11011	MMI2 001001
6	5	5	5	5	6

C790

Format: PREVH rd, rt

Purpose: To reverse halfwords.

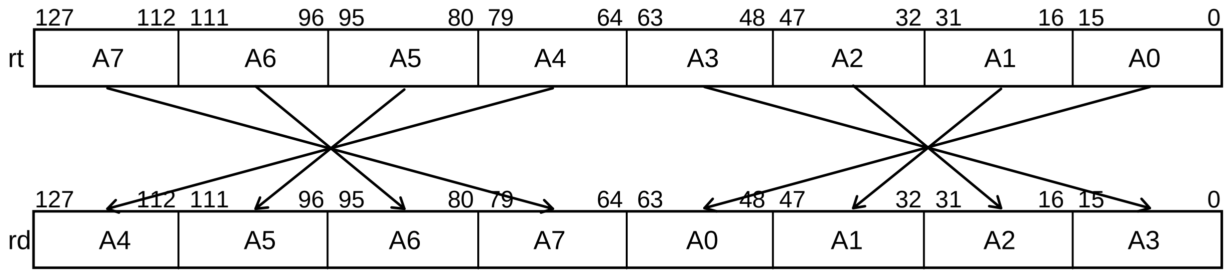
Description: rd ← reverse (rt)

The four high-order halfwords in GPR *rt* are reversed and the four low-order halfwords in GPR *rt* are reversed. The results are placed into GPR *rd.*

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{15..0} \leftarrow GPR[rt]_{63..48}$
$GPR[rd]_{31..16} \leftarrow GPR[rt]_{47..32}$
$GPR[rd]_{47..32} \leftarrow GPR[rt]_{31..16}$
$GPR[rd]_{63..48} \leftarrow GPR[rt]_{15..0}$
$GPR[rd]_{79..64} \leftarrow GPR[rt]_{127..112}$
$GPR[rd]_{95..80} \leftarrow GPR[rt]_{111..96}$
$GPR[rd]_{111..96} \leftarrow GPR[rt]_{95..80}$
$GPR[rd]_{127..112} \leftarrow GPR[rt]_{79..64}$



Exceptions:

None

PROT3W

Parallel Rotate 3 Words Left

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	0 00000	rt	rd	PROT3W 11111	MMI2 001001
6	5	5	5	5	6

C790

Format: PROT3W rd, rt

Purpose: To rotate words.

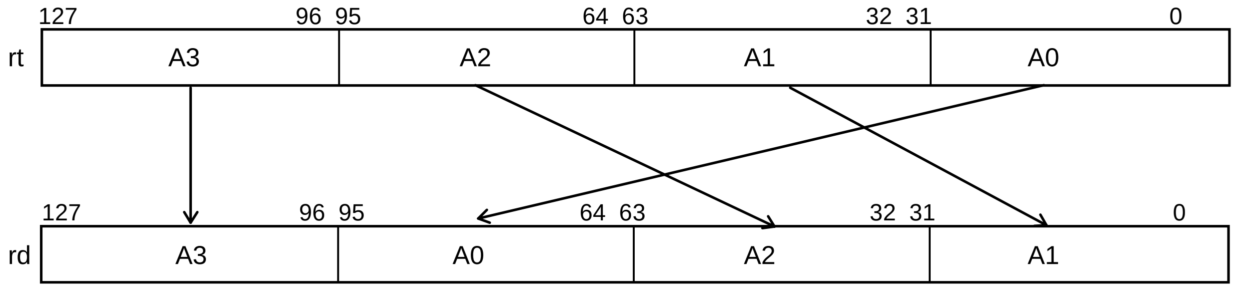
Description: rd ← rotate (rt)

The three low-order words in GPR *rt* are rotated to the right. The results are placed into GPR *rd* while the other word is copied directly to the corresponding word in GPR *rd*.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{31..0} \leftarrow GPR[rt]_{63..32}$
$GPR[rd]_{63..32} \leftarrow GPR[rt]_{95..64}$
$GPR[rd]_{95..64} \leftarrow GPR[rt]_{31..0}$
$GPR[rd]_{127..96} \leftarrow GPR[rt]_{127..96}$



Exceptions:

None

PSLLH

Parallel Shift Left Logical Halfword

PSLLH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	0 00000	rt	rd	sa	PSLLH 110100
6	5	5	5	5	6

C790

Format: PSLLH rd, rt, sa

Purpose: To logically shift left 8 halfwords by a fixed number of bits, in parallel.

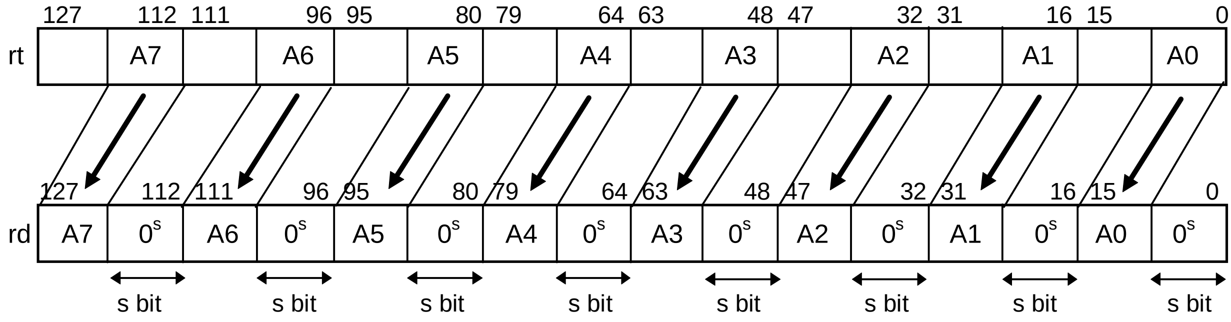
Description: rd ← rt << sa (logical)

The eight halfwords in GPR *rt* are shifted left in parallel, inserting zeros into the emptied bits; the results are placed into the corresponding eight halfwords in GPR *rd*. The bit shift count is specified by the low-order four bits of sa.

This instruction operates on 128-bit registers.

Operation:

$s \leftarrow sa_{3..0}$
$GPR[rd]_{15..0} \leftarrow GPR[rt]_{(15-s)..0} \,||\, 0^s$
$GPR[rd]_{31..16} \leftarrow GPR[rt]_{(31-s)..16} \,||\, 0^s$
$GPR[rd]_{47..32} \leftarrow GPR[rt]_{(47-s)..32} \,||\, 0^s$
$GPR[rd]_{63..48} \leftarrow GPR[rt]_{(63-s)..48} \,||\, 0^s$
$GPR[rd]_{79..64} \leftarrow GPR[rt]_{(79-s)..64} \,||\, 0^s$
$GPR[rd]_{95..80} \leftarrow GPR[rt]_{(95-s)..80} \,||\, 0^s$
$GPR[rd]_{111..96} \leftarrow GPR[rt]_{(111-s)..96} \,||\, 0^s$
$GPR[rd]_{127..112} \leftarrow GPR[rt]_{(127-s)..112} \,||\, 0^s$



Exceptions:

None

PSLLVW Parallel Shift Left Logical Variable Word PSLLVW

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PSLLVW 00010	MMI2 001001
6	5	5	5	5	6

C790

Format: PSLLVW rd, rt, rs

Purpose: To logically shift left 2 words by a variable number of bits, in parallel.

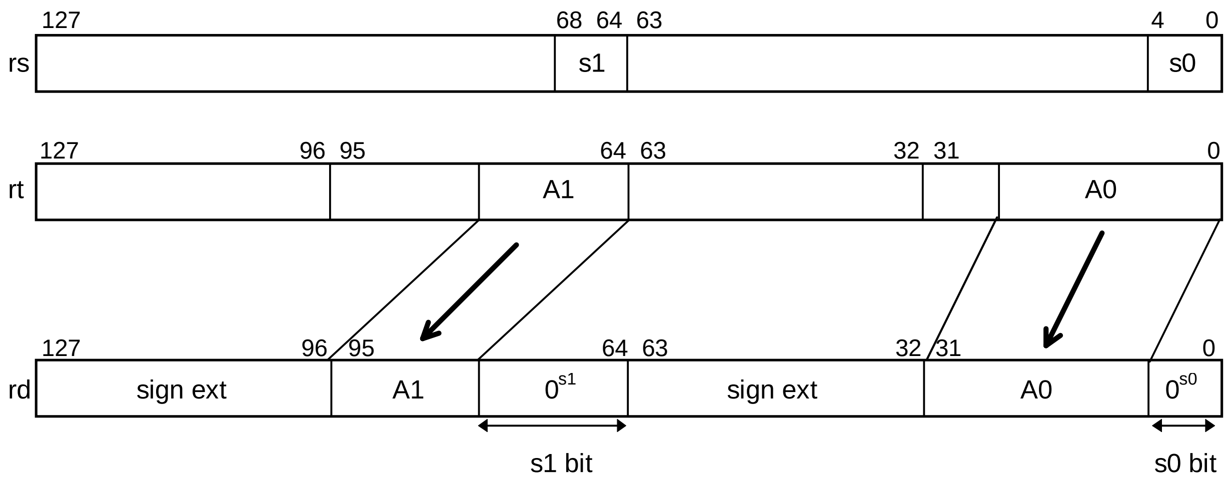
Description: rd ← rt << rs (logical)

The low-order words of the two doublewords in GPR *rt* are shifted left in parallel, inserting zeros into the emptied bits; the results are placed into the corresponding two words in GPR *rd*. The bit shift counts are specified by the low-order five bits of the two doublewords in GPR rs.

This instruction operates on 128-bit registers.

Operation:

$s0 \leftarrow GPR[rs]_{4..0}$
$s1 \leftarrow GPR[rs]_{68..64}$
$temp0 \leftarrow GPR[rt]_{(31-s0)..0} \, || \, 0^{s0}$
$temp1 \leftarrow GPR[rt]_{(95-s1)..64} \, || \, 0^{s1}$
$GPR[rd]_{63..0} \leftarrow (temp0_{31})^{32} \, || \, temp0_{31..0}$
$GPR[rd]_{127..64} \leftarrow (temp1_{31})^{32} \, || \, temp1_{31..0}$



Exceptions:

None

PSLLW Parallel Shift Left Logical Word PSLLW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	0 00000	rt	rd	sa	PSLLW 111100
6	5	5	5	5	6

C790

Format: PSLLW rd, rt, sa

Purpose: To logically shift left 4 words by a fixed number of bits, in parallel.

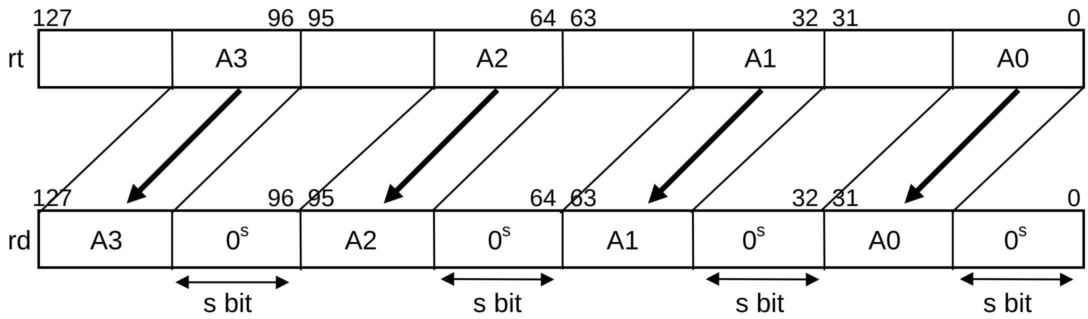
Description: rd ← rt << sa (logical)

The four words in GPR *rt* are shifted left by five bits of *sa* in parallel, inserting zeros into the emptied bits; the results are placed into the corresponding four words in GPR *rd*.

This instruction operates on 128-bit registers.

Operation:

$s \leftarrow sa_{4..0}$
$GPR[rd]_{31..0} \leftarrow GPR[rt]_{(31-s)..0} \parallel 0^s$
$GPR[rd]_{63..32} \leftarrow GPR[rt]_{(63-s)..32} \parallel 0^s$
$GPR[rd]_{95..64} \leftarrow GPR[rt]_{(95-s)..64} \parallel 0^s$
$GPR[rd]_{127..96} \leftarrow GPR[rt]_{(127-s)..96} \parallel 0^s$



Exceptions:

None

PSRAH Parallel Shift Right Arithmetic Halfword PSRAH

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	0 00000	rt	rd	sa	PSRAH 110111
6	5	5	5	5	6

C790

Format: PSRAH rd, rt, sa

Purpose: To arithmetically shift right 8 halfwords by a fixed number of bits, in parallel.

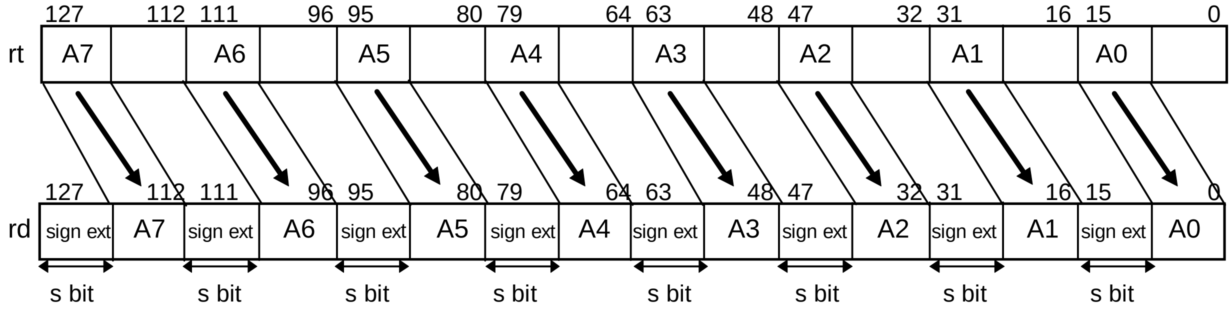
Description: rd ← rt >> sa (arithmetic)

The eight halfwords in GPR *rt* are shifted right by *sa* bits in parallel sign extending the high order bits; the results are placed into the corresponding eight halfwords in GPR *rd*. The bit shift count is specified by the low-order four bits of sa.

This instruction operates on 128-bit registers.

Operation:

$s \leftarrow sa_{3..0}$
$GPR[rd]_{15..0} \leftarrow (GPR[rt]_{15})^{s} \| GPR[rt]_{15..s}$
$GPR[rd]_{31..16} \leftarrow (GPR[rt]_{31})^{s} \| GPR[rt]_{31..(16+s)}$
$GPR[rd]_{47..32} \leftarrow (GPR[rt]_{47})^{s} \| GPR[rt]_{47..(32+s)}$
$GPR[rd]_{63..48} \leftarrow (GPR[rt]_{63})^{s} \| GPR[rt]_{63..(48+s)}$
$GPR[rd]_{79..64} \leftarrow (GPR[rt]_{79})^{s} \| GPR[rt]_{79..(64+s)}$
$GPR[rd]_{95..80} \leftarrow (GPR[rt]_{95})^{s} \| GPR[rt]_{95..(80+s)}$
$GPR[rd]_{111..96} \leftarrow (GPR[rt]_{111})^{s} \| GPR[rt]_{111..(96+s)}$
$GPR[rd]_{127..112} \leftarrow (GPR[rt]_{127})^{s} \| GPR[rt]_{127..(112+s)}$



Exceptions:

None

PSRAVW Parallel Shift Right Arithmetic Variable Word PSRAVW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PSRAVW 00011	MMI3 101001
6	5	5	5	5	6

C790

Format: PSRAVW rd, rt, rs

Purpose: To arithmetically shift right 2 words by a variable number of bits, in parallel.

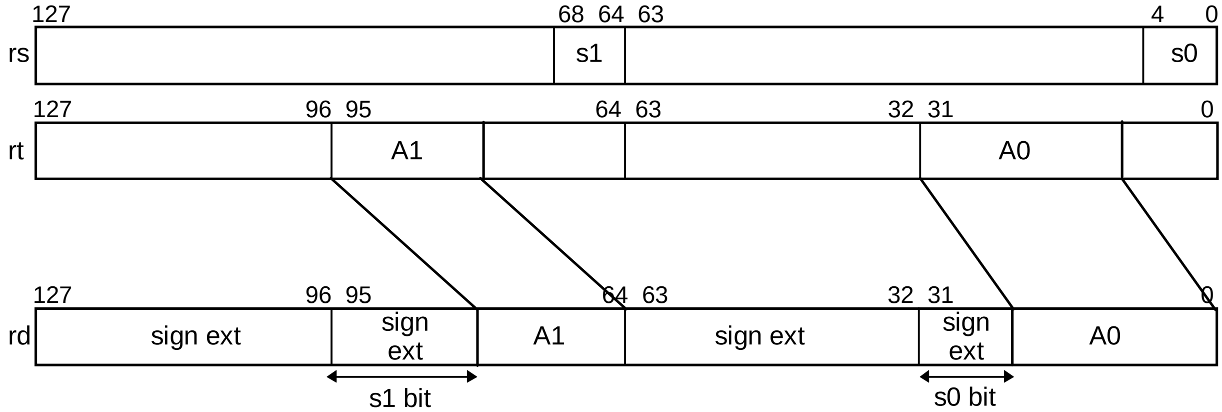
Description: rd ← rt >> rs (arithmetic)

The low-order words of the two doublewords in GPR *rt* are shifted right in parallel, sign extending the high order bits; the results are placed into the corresponding two words in GPR *rd*. The bit shift counts are specified by the low-order five bits of the two doublewords in GPR *rs*.

This instruction operates on 128-bit registers.

Operation:

$s0 \leftarrow GPR[rs]_{4..0}$
$s1 \leftarrow GPR[rs]_{68..64}$
$temp0 \leftarrow (GPR[rt]_{31})^{s0} \parallel GPR[rt]_{31..s0}$
$temp1 \leftarrow (GPR[rt]_{95})^{s1} \parallel GPR[rt]_{95..(64+s1)}$
$GPR[rd]_{63..0} \leftarrow (temp0_{31})^{32} \parallel temp0_{31..0}$
$GPR[rd]_{127..64} \leftarrow (temp1_{31})^{32} \parallel temp1_{31..0}$



Exceptions:

None

PSRAW Parallel Shift Right Arithmetic Word PSRAW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	0 00000	rt	rd	sa	PSRAW 111111
6	5	5	5	5	6

C790

Format: PSRAW rd, rt, sa

Purpose: To arithmetically shift right 4 word by a fixed number of bits, in parallel.

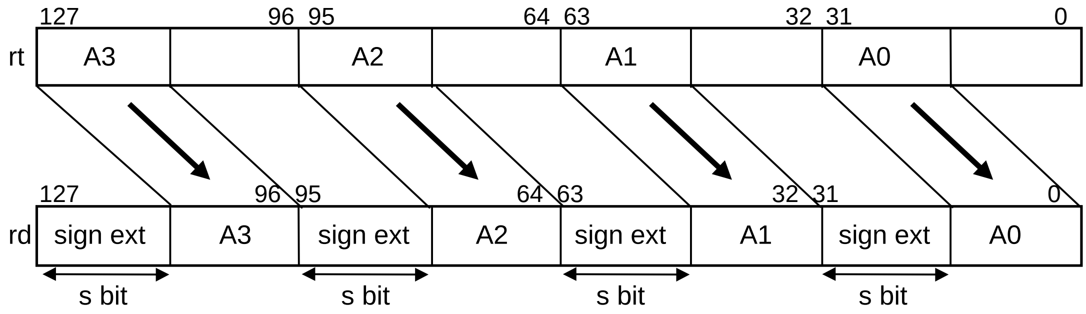
Description: rd ← rt >> sa (arithmetic)

The four words in GPR *rt* are shifted right by five bits of *sa* in parallel, sign extending the high order bits; the results are placed into the corresponding four words in GPR *rd.*

This instruction operates on 128-bit registers.

Operation:

$s \leftarrow sa_{4..0}$
$GPR[rd]_{31..0} \leftarrow (GPR[rt]_{31})^{s} \parallel GPR[rt]_{31..s}$
$GPR[rd]_{63..32} \leftarrow (GPR[rt]_{63})^{s} \parallel GPR[rt]_{63..(32+s)}$
$GPR[rd]_{95..64} \leftarrow (GPR[rt]_{95})^{s} \parallel GPR[rt]_{95..(64+s)}$
$GPR[rd]_{127..96} \leftarrow (GPR[rt]_{127})^{s} \parallel GPR[rt]_{127..(96+s)}$



Exceptions:

None

PSRLH Parallel Shift Right Logical Halfword PSRLH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	0 00000	rt	rd	sa	PSRLH 110110
6	5	5	5	5	6

C790

Format: PSRLH rd, rt, sa

Purpose: To logically shift right 8 halfwords by a fixed number of bits, in parallel.

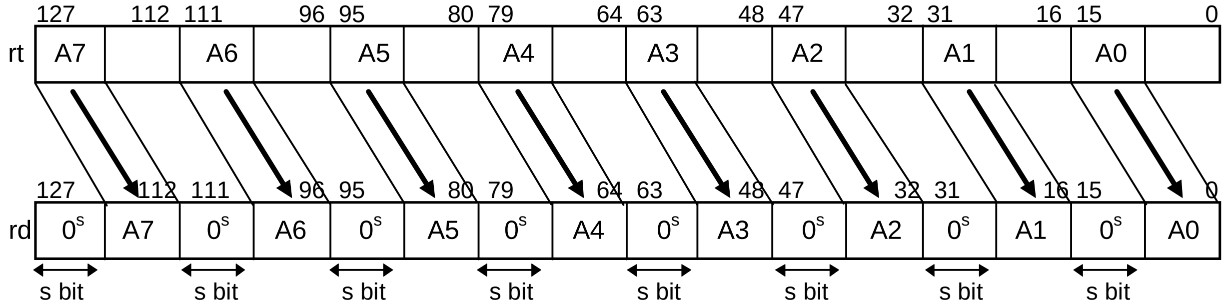
Description: rd ← rt >> sa (logical)

The eight halfwords in GPR *rt* are shifted right by *sa* bits, in parallel, inserting zeros into the high order bits; the results are placed into the corresponding eight halfwords in GPR *rd*. The bit shift count is specified by the low-order four bits of sa.

This instruction operates on 128-bit registers.

Operation:

$s \leftarrow sa_{3..0}$
$GPR[rd]_{15..0} \leftarrow 0^s \,||\, GPR[rt]_{15..s}$
$GPR[rd]_{31..16} \leftarrow 0^s \,||\, GPR[rt]_{31..(16+s)}$
$GPR[rd]_{47..32} \leftarrow 0^s \,||\, GPR[rt]_{47..(32+s)}$
$GPR[rd]_{63..48} \leftarrow 0^s \,||\, GPR[rt]_{63..(48+s)}$
$GPR[rd]_{79..64} \leftarrow 0^s \,||\, GPR[rt]_{79..(64+s)}$
$GPR[rd]_{95..80} \leftarrow 0^s \,||\, GPR[rt]_{95..(80+s)}$
$GPR[rd]_{111..96} \leftarrow 0^s \,||\, GPR[rt]_{111..(96+s)}$
$GPR[rd]_{127..112} \leftarrow 0^s \,||\, GPR[rt]_{127..(112+s)}$



Exceptions:

None

PSRLVW Parallel Shift Right Logical Variable Word PSRLVW

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PSRLVW 00011	MMI2 001001
6	5	5	5	5	6

C790

Format: PSRLVW rd, rt, rs

Purpose: To logically shift right 2 words by a variable number of bits, in parallel.

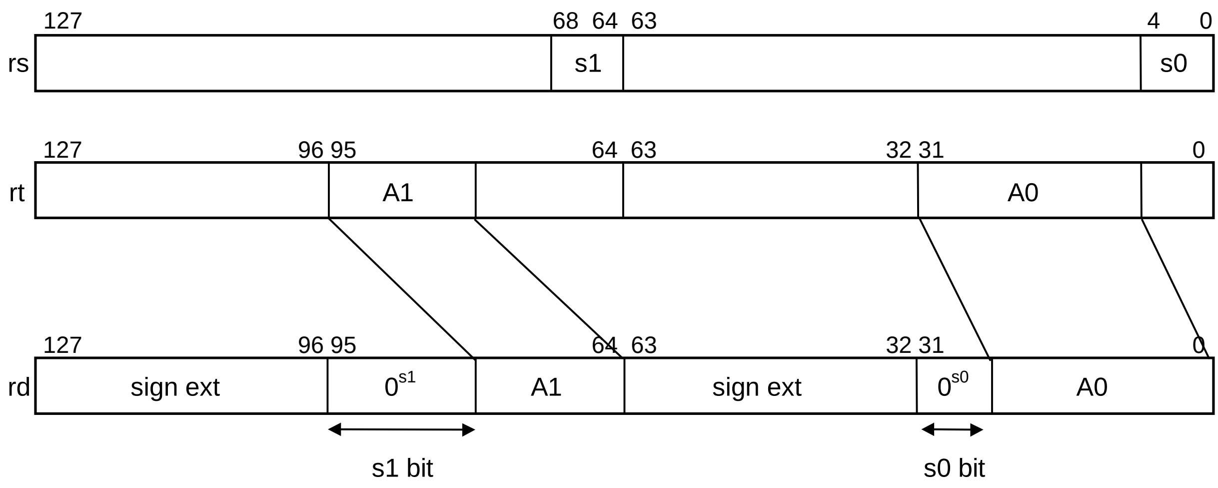
Description: rd ← rt >> rs (logical)

The low-order words of the two doublewords in GPR *rt* are shifted right in parallel, inserting zeros into the high order bits. The results are sign extended; the results are placed into the corresponding two words in GPR *rd*. The bit shift counts are specified by the low-order five bits of the two doublewords in GPR *rs*.

This instruction operates on 128-bit registers.

Operation:

$s0 \leftarrow GPR[rs]_{4..0}$
$s1 \leftarrow GPR[rs]_{68..64}$
$temp0 \leftarrow 0^{s0} \,||\, GPR[rt]_{31..s0}$
$temp1 \leftarrow 0^{s1} \,||\, GPR[rt]_{95..(64+s1)}$
$GPR[rd]_{63..0} \leftarrow (temp0_{31})^{32} \,||\, temp0_{31..0}$
$GPR[rd]_{127..64} \leftarrow (temp1_{31})^{32} \,||\, temp1_{31..0}$



Exceptions:

None

PSRLW — Parallel Shift Right Logical Word — PSRLW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	0 00000	rt	rd	sa	PSRLW 111110
6	5	5	5	5	6

C790

Format: PSRLW rd, rt, sa

Purpose: To logically shift right 4 words by a fixed number of bits, in parallel.

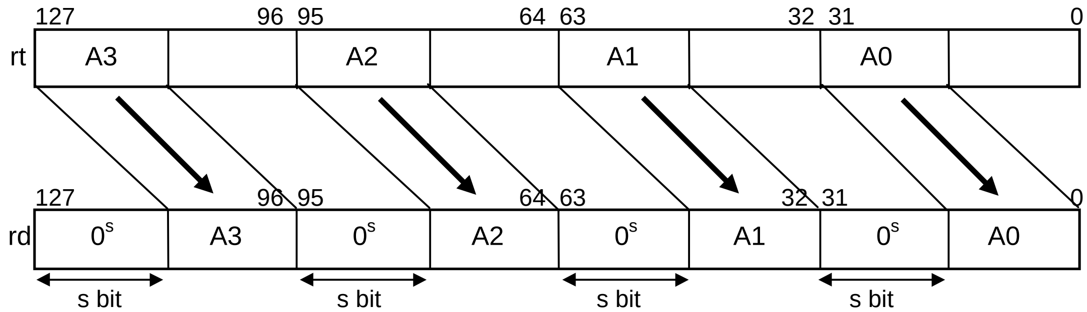
Description: rd ← rt >> sa (logical)

The four words in GPR *rt* are shifted right by five bits of *sa*, in parallel, inserting zeros into the high order bits; the results are placed into the corresponding four words in GPR *rd*.

This instruction operates on 128-bit registers.

Operation:

$s \leftarrow sa_{4..0}$
$GPR[rd]_{31..0} \leftarrow 0^s \,||\, GPR[rt]_{31..s}$
$GPR[rd]_{63..32} \leftarrow 0^s \,||\, GPR[rt]_{63..(32+s)}$
$GPR[rd]_{95..64} \leftarrow 0^s \,||\, GPR[rt]_{95..(64+s)}$
$GPR[rd]_{127..96} \leftarrow 0^s \,||\, GPR[rt]_{127..(96+s)}$



Exceptions:

None

PSUBB Parallel Subtract Byte PSUBB

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PSUBB 01001	MMI0 001000
6	5	5	5	5	6

C790

Format: PSUBB rd, rs, rt

Purpose: To subtract 16 pairs of 8-bit integers in parallel.

Description: rd ← rs – rt

The sixteen signed byte values in GPR *rt* are subtracted from the corresponding sixteen byte values in GPR *rs* in parallel. The results are placed into the corresponding sixteen bytes in GPR *rd.*

No overflow or underflow exceptions are generated under any circumstances.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{7..0} \leftarrow (GPR[rs]_{7..0} - GPR[rt]_{7..0})_{7..0}$
$GPR[rd]_{15..8} \leftarrow (GPR[rs]_{15..8} - GPR[rt]_{15..8})_{7..0}$
$GPR[rd]_{23..16} \leftarrow (GPR[rs]_{23..16} - GPR[rt]_{23..16})_{7..0}$
$GPR[rd]_{31..24} \leftarrow (GPR[rs]_{31..24} - GPR[rt]_{31..24})_{7..0}$
$GPR[rd]_{39..32} \leftarrow (GPR[rs]_{39..32} - GPR[rt]_{39..32})_{7..0}$
$GPR[rd]_{47..40} \leftarrow (GPR[rs]_{47..40} - GPR[rt]_{47..40})_{7..0}$
$GPR[rd]_{55..48} \leftarrow (GPR[rs]_{55..48} - GPR[rt]_{55..48})_{7..0}$
$GPR[rd]_{63..56} \leftarrow (GPR[rs]_{63..56} - GPR[rt]_{63..56})_{7..0}$
$GPR[rd]_{71..64} \leftarrow (GPR[rs]_{71..64} - GPR[rt]_{71..64})_{7..0}$
$GPR[rd]_{79..72} \leftarrow (GPR[rs]_{79..72} - GPR[rt]_{79..72})_{7..0}$
$GPR[rd]_{87..80} \leftarrow (GPR[rs]_{87..80} - GPR[rt]_{87..80})_{7..0}$
$GPR[rd]_{95..88} \leftarrow (GPR[rs]_{95..88} - GPR[rt]_{95..88})_{7..0}$
$GPR[rd]_{103..96} \leftarrow (GPR[rs]_{103..96} - GPR[rt]_{103..96})_{7..0}$
$GPR[rd]_{111..104} \leftarrow (GPR[rs]_{111..104} - GPR[rt]_{111..104})_{7..0}$
$GPR[rd]_{119..112} \leftarrow (GPR[rs]_{119..112} - GPR[rt]_{119..112})_{7..0}$
$GPR[rd]_{127..120} \leftarrow (GPR[rs]_{127..120} - GPR[rt]_{127..120})_{7..0}$

	127 120	119 112	111 104	103 96	95 88	87 80	79 72	71 64	63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
rs	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
rt	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

	127 120	119 112	111 104	103 96	95 88	87 80	79 72	71 64	63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
rd	A15 – B15	A14 – B14	A13 – B13	A12 – B12	A11 – B11	A10 – B10	A9 – B9	A8 – B8	A7 – B7	A6 – B6	A5 – B5	A4 – B4	A3 – B3	A2 – B2	A1 – B1	A0 – B0

Exceptions:

None

PSUBH Parallel Subtract Halfword PSUBH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PSUBH 00101	MMI0 001000
6	5	5	5	5	6

C790

Format: PSUBH rd, rs, rt

Purpose: To subtract 8 pairs of 16-bit integers in parallel.

Description: rd ← rs – rt

The eight signed halfwords in GPR *rt* are subtracted from the corresponding eight halfwords in GPR *rs* in parallel. The results are placed into the corresponding eight halfwords in GPR *rd.*

No overflow or underflow exceptions are generated under any circumstances.

This instruction operates on 128-bit registers.

Operation:

$$GPR[rd]_{15..0} \leftarrow (GPR[rs]_{15..0} - GPR[rt]_{15..0})_{15..0}$$
$$GPR[rd]_{31..16} \leftarrow (GPR[rs]_{31..16} - GPR[rt]_{31..16})_{15..0}$$
$$GPR[rd]_{47..32} \leftarrow (GPR[rs]_{47..32} - GPR[rt]_{47..32})_{15..0}$$
$$GPR[rd]_{63..48} \leftarrow (GPR[rs]_{63..48} - GPR[rt]_{63..48})_{15..0}$$
$$GPR[rd]_{79..64} \leftarrow (GPR[rs]_{79..64} - GPR[rt]_{79..64})_{15..0}$$
$$GPR[rd]_{95..80} \leftarrow (GPR[rs]_{95..80} - GPR[rt]_{95..80})_{15..0}$$
$$GPR[rd]_{111..96} \leftarrow (GPR[rs]_{111..96} - GPR[rt]_{111..96})_{15..0}$$
$$GPR[rd]_{127..112} \leftarrow (GPR[rs]_{127..112} - GPR[rt]_{127..112})_{15..0}$$

	127–112	111–96	95–80	79–64	63–48	47–32	31–16	15–0
rs	A7	A6	A5	A4	A3	A2	A1	A0
	–	–	–	–	–	–	–	–
rt	B7	B6	B5	B4	B3	B2	B1	B0
rd	A7–B7	A6–B6	A5–B5	A4–B4	A3–B3	A2–B2	A1–B1	A0–B0

Exceptions:

None

PSUBSB Parallel Subtract with Signed saturation Byte PSUBSB

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PSUBSB 11001	MMI0 001000
6	5	5	5	5	6

C790

Format: PSUBSB rd, rs, rt

Purpose: To subtract 16 pairs of 8-bit signed integers with saturation in parallel.

Description: rd ← rs – rt

The sixteen signed bytes in GPR *rt* are subtracted from the corresponding sixteen signed bytes in GPR *rs* in parallel. The results are placed into the corresponding sixteen bytes in GPR *rd.*

No overflow or underflow exceptions are generated under any circumstances. Results beyond the range of a signed byte value are saturated according to the following:

Overflow: 0x7F

Underflow: 0x80

This instruction operates on 128-bit registers.

Operation:

```
if ((GPR[rs]7..0 – GPR[rt]7..0) > 0x7F) then
  GPR[rd]7..0  ← 0x7F
else if (0x100 <= (GPR[rs]7..0 – GPR[rt]7..0) < 0x180) then
  GPR[rd]7..0  ← 0x80
else
  GPR[rd]7..0  ← (GPR[rs]7..0 – GPR[rt]7..0)7..0
endif

if ((GPR[rs]15..8 – GPR[rt]15..8) > 0x7F) then
  GPR[rd]15..8        ← 0x7F
else if (0x100 <= (GPR[rs]15..8 – GPR[rt]15..8) < 0x180) then
  GPR[rd]15..8        ← 0x80
else
  GPR[rd]15..8        ← (GPR[rs]15..8 – GPR[rt]15..8)7..0
endif

if ((GPR[rs]23..16 – GPR[rt]23..16) > 0x7F) then
  GPR[rd]23..16       ← 0x7F
else if (0x100 <= (GPR[rs]23..16 – GPR[rt]23..16) < 0x180) then
  GPR[rd]23..16       ← 0x80
else
  GPR[rd]23..16       ← (GPR[rs]23..16 – GPR[rt]23..16)7..0
endif
```

```
if ((GPR[rs]31..24 – GPR[rt]31..24) > 0x7F) then
  GPR[rd]31..24          ← 0x7F
else if (0x100 <= (GPR[rs]31..24 – GPR[rt]31..24) < 0x180) then
  GPR[rd]31..24          ← 0x80
else
  GPR[rd]31..24          ← (GPR[rs]31..24 – GPR[rt]31..24)7..0
endif

if ((GPR[rs]39..32 – GPR[rt]39..32) > 0x7F) then
  GPR[rd]39..32          ← 0x7F
else if (0x100 <= (GPR[rs]39..32 – GPR[rt]39..32) < 0x180) then
  GPR[rd]39..32          ← 0x80
else
  GPR[rd]39..32          ← (GPR[rs]39..32 – GPR[rt]39..32)7..0
endif

if ((GPR[rs]47..40 – GPR[rt]47..40) > 0x7F) then
  GPR[rd]47..40          ← 0x7F
else if (0x100 <= (GPR[rs]47..40 – GPR[rt]47..40) < 0x180) then
  GPR[rd]47..40          ← 0x80
else
  GPR[rd]47..40          ← (GPR[rs]47..40 – GPR[rt]47..40)7..0
endif

if ((GPR[rs]55..48 – GPR[rt]55..48) > 0x7F) then
  GPR[rd]55..48          ← 0x7F
else if (0x100 <= (GPR[rs]55..48 – GPR[rt]55..48) < 0x180) then
  GPR[rd]55..48          ← 0x80
else
  GPR[rd]55..48          ← (GPR[rs]55..48 – GPR[rt]55..48)7..0
endif

if ((GPR[rs]63..56 – GPR[rt]63..56) > 0x7F) then
  GPR[rd]63..56          ← 0x7F
else if (0x100 <= (GPR[rs]63..56 – GPR[rt]63..56) < 0x180) then
  GPR[rd]63..56          ← 0x80
else
  GPR[rd]63..56          ← (GPR[rs]63..56 – GPR[rt]63..56)7..0
endif

if ((GPR[rs]71..64 – GPR[rt]71..64) > 0x7F) then
  GPR[rd]71..64          ← 0x7F
else if (0x100 <= (GPR[rs]71..64 – GPR[rt]71..64) < 0x180) then
  GPR[rd]71..64          ← 0x80
else
  GPR[rd]71..64          ← (GPR[rs]71..64 – GPR[rt]71..64)7..0
endif
```

```
if ((GPR[rs]79..72 – GPR[rt]79..72) > 0x7F) then
  GPR[rd]79..72         ← 0x7F
else if (0x100 <= (GPR[rs]79..72 – GPR[rt]79..72) < 0x180) then
  GPR[rd]79..72         ← 0x80
else
  GPR[rd]79..72         ← (GPR[rs]79..72 – GPR[rt]79..72)7..0
endif

if ((GPR[rs]87..80 – GPR[rt]87..80) > 0x7F) then
  GPR[rd]87..80         ← 0x7F
else if (0x100 <= (GPR[rs]87..80 – GPR[rt]87..80) < 0x180) then
  GPR[rd]87..80         ← 0x80
else
  GPR[rd]87..80         ← (GPR[rs]87..80 – GPR[rt]87..80)7..0
endif

if ((GPR[rs]95..88 – GPR[rt]95..88) > 0x7F) then
  GPR[rd]95..88         ← 0x7F
else if (0x100 <= (GPR[rs]95..88 – GPR[rt]95..88) < 0x180) then
  GPR[rd]95..88         ← 0x80
else
  GPR[rd]95..88         ← (GPR[rs]95..88 – GPR[rt]95..88)7..0
endif

if ((GPR[rs]103..96 – GPR[rt]103..96) > 0x7F) then
  GPR[rd]103..96        ← 0x7F
else if (0x100 <= (GPR[rs]103..96 – GPR[rt]103..96) < 0x180) then
  GPR[rd]103..96        ← 0x80
else
  GPR[rd]103..96        ← (GPR[rs]103..96 – GPR[rt]103..96)7..0
endif

if ((GPR[rs]111..104 – GPR[rt]111..104) > 0x7F) then
  GPR[rd]111..104       ← 0x7F
else if (0x100 <= (GPR[rs]111..104 – GPR[rt]111..104) < 0x180) then
  GPR[rd]111..104       ← 0x80
else
  GPR[rd]111..104       ← (GPR[rs]111..104 – GPR[rt]111..104)7..0
endif

if ((GPR[rs]119..112 – GPR[rt]119..112) > 0x7F) then
  GPR[rd]119..112       ← 0x7F
else if (0x100 <= (GPR[rs]119..112 – GPR[rt]119..112) < 0x180) then
  GPR[rd]119..112       ← 0x80
else
  GPR[rd]119..112       ← (GPR[rs]119..112 – GPR[rt]119..112)7..0
endif
```

```
if ((GPR[rs]127..120 – GPR[rt]127..120) > 0x7F) then
  GPR[rd]127..120          ← 0x7F
else if (0x100 <= (GPR[rs]127..120 – GPR[rt]127..120) < 0x180) then
  GPR[rd]127..120          ← 0x80
else
  GPR[rd]127..120          ← (GPR[rs]127..120 – GPR[rt]127..120)7..0
endif
```

	127 120	119 112	111 104	103 96	95 88	87 80	79 72	71 64	63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
rs	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
rt	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
rd	A15 – B15	A14 – B14	A13 – B13	A12 – B12	A11 – B11	A10 – B10	A9 – B9	A8 – B8	A7 – B7	A6 – B6	A5 – B5	A4 – B4	A3 – B3	A2 – B2	A1 – B1	A0 – B0

* Saturate to signed byte

Exceptions:

None

PSUBSH Parallel Subtract with Signed Saturation Halfword PSUBSH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PSUBSH 10101	MMI0 001000
6	5	5	5	5	6

C790

Format: PSUBSH rd, rs, rt

Purpose: To subtract 8 pairs of 16-bit signed integers with saturation in parallel.

Description: rd ← rs – rt

The eight signed halfwords in GPR *rt* are subtracted from the corresponding eight signed halfwords in GPR *rs* in parallel. The results are placed into the corresponding eight halfwords in GPR *rd*.

No overflow or underflow exceptions are generated under any circumstances. Results beyond the range of a signed halfword value are saturated according to the following:

Overflow: 0x7FFF

Underflow: 0x8000

This instruction operates on 128-bit registers.

Operation:

```
if ((GPR[rs]15..0 – GPR[rt]15..0) > 0x7FFF) then
   GPR[rd]15..0          ← 0x7FFF
else if (0x10000 <= (GPR[rs]15..0 – GPR[rt]15..0) < 0x18000) then
   GPR[rd]15..0          ← 0x8000
else
   GPR[rd]15..0          ← (GPR[rs]15..0 – GPR[rt]15..0)15..0
endif

if ((GPR[rs]31..16 – GPR[rt]31..16) > 0x7FFF) then
   GPR[rd]31..16         ← 0x7FFF
else if (0x10000 <= (GPR[rs]31..16 – GPR[rt]31..16) < 0x18000) then
   GPR[rd]31..16         ← 0x8000
else
   GPR[rd]31..16         ← (GPR[rs]31..16 – GPR[rt]31..16)15..0
endif

if ((GPR[rs]47..32 – GPR[rt]47..32) > 0x7FFF) then
   GPR[rd]47..32         ← 0x7FFF
else if (0x10000 <= (GPR[rs]47..32 – GPR[rt]47..32) < 0x18000) then
   GPR[rd]47..32         ← 0x8000
else
   GPR[rd]47..32         ← (GPR[rs]47..32 – GPR[rt]47..32)15..0
endif

if ((GPR[rs]63..48 – GPR[rt]63..48) > 0x7FFF) then
   GPR[rd]63..48         ← 0x7FFF
else if (0x10000 <= (GPR[rs]63..48 – GPR[rt]63..48) < 0x18000) then
```

GPR[rd]$_{63..48}$ ← 0x8000
else
GPR[rd]$_{63..48}$ ← (GPR[rs]$_{63..48}$ – GPR[rt]$_{63..48}$)$_{15..0}$
endif

if ((GPR[rs]$_{79..64}$ – GPR[rt]$_{79..64}$) > 0x7FFF) then
GPR[rd]$_{79..64}$ ← 0x7FFF
else if (0x10000 <= (GPR[rs]$_{79..64}$ – GPR[rt]$_{79..64}$) < 0x18000) then
GPR[rd]$_{79..64}$ ← 0x8000
else
GPR[rd]$_{79..64}$ ← (GPR[rs]$_{79..64}$ – GPR[rt]$_{79..64}$)$_{15..0}$
endif

if ((GPR[rs]$_{95..80}$ – GPR[rt]$_{95..80}$) > 0x7FFF) then
GPR[rd]$_{95..80}$ ← 0x7FFF
else if (0x10000 <= (GPR[rs]$_{95..80}$ – GPR[rt]$_{95..80}$) < 0x18000) then
GPR[rd]$_{95..80}$ ← 0x8000
else
GPR[rd]$_{95..80}$ ← (GPR[rs]$_{95..80}$ – GPR[rt]$_{95..80}$)$_{15..0}$
endif

if ((GPR[rs]$_{111..96}$ – GPR[rt]$_{111..96}$) > 0x7FFF) then
GPR[rd]$_{111..96}$ ← 0x7FFF
else if (0x10000 <= (GPR[rs]$_{111..96}$ – GPR[rt]$_{111..96}$) < 0x18000) then
GPR[rd]$_{111..96}$ ← 0x8000
else
GPR[rd]$_{111..96}$ ← (GPR[rs]$_{111..96}$ – GPR[rt]$_{111..96}$)$_{15..0}$
endif

if ((GPR[rs]$_{127..112}$ – GPR[rt]$_{127..112}$) > 0x7FFF) then
GPR[rd]$_{127..112}$ ← 0x7FFF
else if (0x10000 <= (GPR[rs]$_{127..112}$ – GPR[rt]$_{127..112}$) < 0x18000) then
GPR[rd]$_{127..112}$ ← 0x8000
else
GPR[rd]$_{127..112}$ ← (GPR[rs]$_{127..112}$ – GPR[rt]$_{127..112}$)$_{15..0}$
endif

	127 112	111 96	95 80	79 64	63 48	47 32	31 16	15 0
rs	A7	A6	A5	A4	A3	A2	A1	A0
	–	–	–	–	–	–	–	–
rt	B7	B6	B5	B4	B3	B2	B1	B0
rd	A7–B7	A6–B6	A5–B5	A4–B4	A3–B3	A2–B2	A1–B1	A0–B0

* Saturate to signed halfword

Exceptions:

None

PSUBSW Parallel Subtract with Signed Saturation Word PSUBSW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PSUBSW 10001	MMI0 001000
6	5	5	5	5	6

C790

Format: PSUBSW rd, rs, rt

Purpose: To subtract 4 pairs of 32-bit signed integers with saturation in parallel.

Description: rd ← rs – rt

The four signed words in GPR *rt* are subtracted from the corresponding four signed words in GPR *rs* in parallel. The results are placed into the corresponding four words in GPR *rd*.

No overflow or underflow exceptions are generated under any circumstances. Results beyond the range of a signed word value are saturated according to the following:

Overflow: 0x7FFFFFFF

Underflow: 0x80000000

This instruction operates on 128-bit registers.

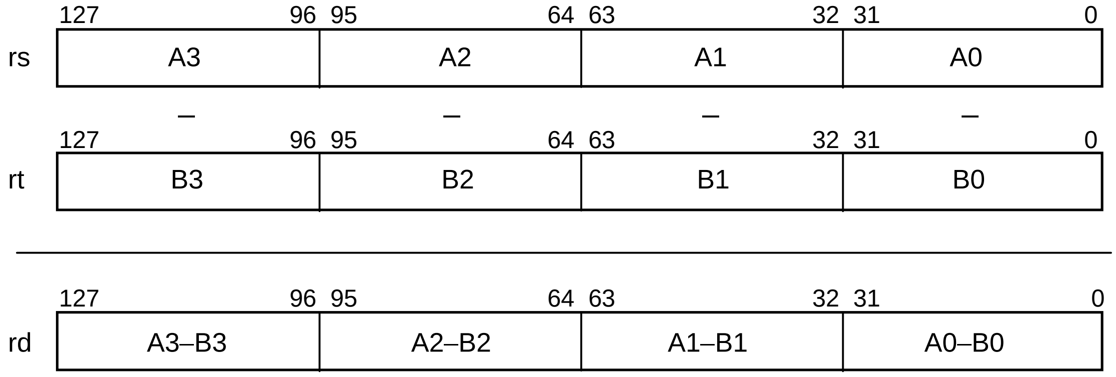
Operation:

```
if ((GPR[rs]31..0 – GPR[rt]31..0) > 0x7FFFFFFF) then
  GPR[rd]31..0        ← 0x7FFFFFFF
else if (0x100000000 <= (GPR[rs]31..0 – GPR[rt]31..0) < 0x180000000) then
  GPR[rd]31..0        ← 0x80000000
else
  GPR[rd]31..0        ← (GPR[rs]31..0 – GPR[rt]31..0)31..0
endif

if ((GPR[rs]63..32 – GPR[rt]63..32) > 0x7FFFFFFF) then
  GPR[rd]63..32       ← 0x7FFFFFFF
else if (0x100000000 <= (GPR[rs]63..32 – GPR[rt]63..32) < 0x180000000) then
  GPR[rd]63..32       ← 0x80000000
else
  GPR[rd]63..32       ← (GPR[rs]63..32 – GPR[rt]63..32)31..0
endif

if ((GPR[rs]95..64 – GPR[rt]95..64) > 0x7FFFFFFF) then
  GPR[rd]95..64       ← 0x7FFFFFFF
else if (0x100000000 <= (GPR[rs]95..64 – GPR[rt]95..64) < 0x180000000) then
  GPR[rd]95..64       ← 0x80000000
else
  GPR[rd]95..64       ← (GPR[rs]95..64 – GPR[rt]95..64)31..0
endif
```

```
if ((GPR[rs]127..96 – GPR[rt]127..96) > 0x7FFFFFFF) then
  GPR[rd]127..96          ← 0x7FFFFFFF
else if (0x100000000 <= (GPR[rs]127..96 – GPR[rt]127..96) < 0x180000000) then
  GPR[rd]127..96          ← 0x80000000
else
  GPR[rd]127..96          ← (GPR[rs]127..96 – GPR[rt]127..96)31..0
endif
```



* Saturate to signed word

Exceptions:

None

PSUBUB Parallel Subtract with Unsigned Saturation Byte PSUBUB

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PSUBUB 11001	MMI1 101000
6	5	5	5	5	6

C790

Format: PSUBUB rd, rs, rt

Purpose: To subtract 16 pairs of 8-bit unsigned integers with saturation in parallel.

Description: rd ← rs – rt

The sixteen unsigned bytes in GPR *rt* are subtracted from the corresponding sixteen unsigned bytes in GPR *rs* in parallel. The results are placed into the corresponding sixteen bytes in GPR *rd.*

No underflow exceptions are generated under any circumstances. Results beyond the range of an unsigned byte value are saturated according to the following:

Underflow: 0x00

This instruction operates on 128-bit registers.

Operation:

```
if ((GPR[rs]7..0 – GPR[rt]7..0) < 0x00) then
  GPR[rd]7..0 ← 0x00
else
  GPR[rd]7..0 ← (GPR[rs]7..0 – GPR[rt]7..0)7..0
endif

if ((GPR[rs]15..8 – GPR[rt]15..8) < 0x00) then
  GPR[rd]15..8 ← 0x00
else
  GPR[rd]15..8 ← (GPR[rs]15..8 – GPR[rt]15..8)7..0
endif

if ((GPR[rs]23..16 – GPR[rt]23..16) < 0x00) then
  GPR[rd]23..16          ← 0x00
else
  GPR[rd]23..16          ← (GPR[rs]23..16 – GPR[rt]23..16)7..0
endif

if ((GPR[rs]31..24 – GPR[rt]31..24) < 0x00) then
  GPR[rd]31..24          ← 0x00
else
  GPR[rd]31..24          ← (GPR[rs]31..24 – GPR[rt]31..24)7..0
endif

if ((GPR[rs]39..32 – GPR[rt]39..32) < 0x00) then
  GPR[rd]39..32          ← 0x00
else
  GPR[rd]39..32          ← (GPR[rs]39..32 – GPR[rt]39..32)7..0
endif
```

```
if ((GPR[rs]47..40 – GPR[rt]47..40) < 0x00) then
  GPR[rd]47..40          ← 0x00
else
  GPR[rd]47..40          ← (GPR[rs]47..40 – GPR[rt]47..40)7..0
endif

if ((GPR[rs]55..48 – GPR[rt]55..48) < 0x00) then
  GPR[rd]55..48          ← 0x00
else
  GPR[rd]55..48          ← (GPR[rs]55..48 – GPR[rt]55..48)7..0
endif

if ((GPR[rs]63..56 – GPR[rt]63..56) < 0x00) then
  GPR[rd]63..56          ← 0x00
else
  GPR[rd]63..56          ← (GPR[rs]63..56 – GPR[rt]63..56)7..0
endif

if ((GPR[rs]71..64 – GPR[rt]71..64) < 0x00) then
  GPR[rd]71..64          ← 0x00
else
  GPR[rd]71..64          ← (GPR[rs]71..64 – GPR[rt]71..64)7..0
endif

if ((GPR[rs]79..72 – GPR[rt]79..72) < 0x00) then
  GPR[rd]79..72          ← 0x00
else
  GPR[rd]79..72          ← (GPR[rs]79..72 – GPR[rt]79..72)7..0
endif

if ((GPR[rs]87..80 – GPR[rt]87..80) < 0x00) then
  GPR[rd]87..80          ← 0x00
else
  GPR[rd]87..80          ← (GPR[rs]87..80 – GPR[rt]87..80)7..0
endif

if ((GPR[rs]95..88 – GPR[rt]95..88) < 0x00) then
  GPR[rd]95..88          ← 0x00
else
  GPR[rd]95..88          ← (GPR[rs]95..88 – GPR[rt]95..88)7..0
endif

if ((GPR[rs]103..96 – GPR[rt]103..96) < 0x00) then
  GPR[rd]103..96         ← 0x00
else
  GPR[rd]103..96         ← (GPR[rs]103..96 – GPR[rt]103..96)7..0
endif

if ((GPR[rs]111..104 – GPR[rt]111..104) < 0x00) then
  GPR[rd]111..104        ← 0x00
else
  GPR[rd]111..104        ← (GPR[rs]111..104 – GPR[rt]111..104)7..0
endif
```

```
if ((GPR[rs]119..112 – GPR[rt]119..112) < 0x00) then
  GPR[rd]119..112          ← 0x00
else
  GPR[rd]119..112          ← (GPR[rs]119..112 – GPR[rt]119..112)7..0
endif

if ((GPR[rs]127..120 – GPR[rt]127..120) < 0x00) then
  GPR[rd]127..120          ← 0x00
else
  GPR[rd]127..120          ← (GPR[rs]127..120 – GPR[rt]127..120)7..0
endif
```

	127 120	119 112	111 104	103 96	95 88	87 80	79 72	71 64	63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
rs	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
rt	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

	127 120	119 112	111 104	103 96	95 88	87 80	79 72	71 64	63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
rd	A15 – B15	A14 – B14	A13 – B13	A12 – B12	A11 – B11	A10 – B10	A9 – B9	A8 – B8	A7 – B7	A6 – B6	A5 – B5	A4 – B4	A3 – B3	A2 – B2	A1 – B1	A0 – B0

* Saturate to unsigned byte

Exceptions:

None

PSUBUH Parallel Subtract with Unsigned Saturation Halfword PSUBUH

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PSUBUH 10101	MMI1 101000
6	5	5	5	5	6

C790

Format: PSUBUH rd, rs, rt

Purpose: To subtract 8 pairs of 16-bit unsigned integers with saturation in parallel.

Description: rd ← rs – rt

The eight unsigned halfwords in GPR *rt* are subtracted from the corresponding eight unsigned halfwords in GPR *rs* in parallel. The results are placed into the corresponding eight halfwords in GPR *rd.*

No underflow exceptions are generated under any circumstances. Results beyond the range of an unsigned halfword value are saturated according to the following:

Underflow: 0x0000

This instruction operates on 128-bit registers.

Operation:

```
if ((GPR[rs]15..0 – GPR[rt]15..0) < 0x0000) then
  GPR[rd]15..0          ← 0x0000
else
  GPR[rd]15..0          ← (GPR[rs]15..0 – GPR[rt]15..0)15..0
endif

if ((GPR[rs]31..16 – GPR[rt]31..16) < 0x0000) then
  GPR[rd]31..16         ← 0x0000
else
  GPR[rd]31..16         ← (GPR[rs]31..16 – GPR[rt]31..16)15..0
endif

if ((GPR[rs]47..32 – GPR[rt]47..32) < 0x0000) then
  GPR[rd]47..32         ← 0x0000
else
  GPR[rd]47..32         ← (GPR[rs]47..32 – GPR[rt]47..32)15..0
endif

if ((GPR[rs]63..48 – GPR[rt]63..48) < 0x0000) then
  GPR[rd]63..48         ← 0x0000
else
  GPR[rd]63..48         ← (GPR[rs]63..48 – GPR[rt]63..48)15..0
endif

if ((GPR[rs]79..64 – GPR[rt]79..64) < 0x0000) then
  GPR[rd]79..64         ← 0x0000
else
  GPR[rd]79..64         ← (GPR[rs]79..64 – GPR[rt]79..64)15..0
endif
```

if $((GPR[rs]_{95..80} - GPR[rt]_{95..80}) < 0x0000)$ then
  $GPR[rd]_{95..80} \leftarrow 0x0000$
else
  $GPR[rd]_{95..80} \leftarrow (GPR[rs]_{95..80} - GPR[rt]_{95..80})_{15..0}$
endif

if $((GPR[rs]_{111..96} - GPR[rt]_{111..96}) < 0x0000)$ then
  $GPR[rd]_{111..96} \leftarrow 0x0000$
else
  $GPR[rd]_{111..96} \leftarrow (GPR[rs]_{111..96} - GPR[rt]_{111..96})_{15..0}$
endif

if $((GPR[rs]_{127..112} - GPR[rt]_{127..112}) < 0x0000)$ then
  $GPR[rd]_{127..112} \leftarrow 0x0000$
else
  $GPR[rd]_{127..112} \leftarrow (GPR[rs]_{127..112} - GPR[rt]_{127..112})_{15..0}$
endif

	127–112	111–96	95–80	79–64	63–48	47–32	31–16	15–0
rs	A7	A6	A5	A4	A3	A2	A1	A0
	–	–	–	–	–	–	–	–
rt	B7	B6	B5	B4	B3	B2	B1	B0
rd	A7–B7	A6–B6	A5–B5	A4–B4	A3–B3	A2–B2	A1–B1	A0–B0

* Saturate to unsigned halfword

Exceptions:

None

PSUBUW Parallel Subtract with Unsigned Saturation Word PSUBUW

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PSUBUW 10001	MMI1 101000
6	5	5	5	5	6

C790

Format: PSUBUW rd, rs, rt

Purpose: To subtract 4 pairs of 32-bit unsigned integers with saturation in parallel.

Description: rd ← rs – rt

The four unsigned words in GPR *rt* are subtracted from the corresponding four unsigned words in GPR *rs* in parallel. The results are placed into the corresponding four words in GPR *rd.*

No underflow exceptions are generated under any circumstances. Results beyond the range of an unsigned word value are saturated according to the following:

Underflow: 0x00000000

This instruction operates on 128-bit registers.

Operation:

```
if ((GPR[rs]31..0 – GPR[rt]31..0) < 0x00000000) then
  GPR[rd]31..0        ← 0x00000000
else
  GPR[rd]31..0        ← (GPR[rs]31..0 – GPR[rt]31..0)31..0
endif

if ((GPR[rs]63..32 – GPR[rt]63..32) < 0x00000000) then
  GPR[rd]63..32       ← 0x00000000
else
  GPR[rd]63..32       ← (GPR[rs]63..32 – GPR[rt]63..32)31..0
endif

if ((GPR[rs]95..64 – GPR[rt]95..64) < 0x00000000) then
  GPR[rd]95..64       ← 0x00000000
else
  GPR[rd]95..64       ← (GPR[rs]95..64 – GPR[rt]95..64)31..0
endif

if ((GPR[rs]127..96 – GPR[rt]127..96) < 0x00000000) then
  GPR[rd]127..96      ← 0x00000000
else
  GPR[rd]127..96      ← (GPR[rs]127..96 – GPR[rt]127..96)31..0
endif
```

	127 … 96	95 … 64	63 … 32	31 … 0
rs	A3	A2	A1	A0
	–	–	–	–
rt	B3	B2	B1	B0
rd	A3–B3	A2–B2	A1–B1	A0–B0

* Saturate to Unsigned word

Exceptions:

None

PSUBW Parallel Subtract Word PSUBW

31–26	25–21	20–16	15–11	10–6	5–0
MMI 011100	rs	rt	rd	PSUBW 00001	MMI0 001000
6	5	5	5	5	6

C790

Format: PSUBW rd, rs, rt

Purpose: To subtract 4 pairs of 32-bit integers in parallel.

Description: rd ← rs – rt

The four signed words in GPR *rt* are subtracted from the corresponding four words in GPR *rs* in parallel. The results are placed into the corresponding four words in GPR *rd.*

No overflow or underflow exceptions are generated under any circumstances.

This instruction operates on 128-bit registers.

Operation:

$$GPR[rd]_{31..0} \leftarrow (GPR[rs]_{31..0} - GPR[rt]_{31..0})_{31..0}$$
$$GPR[rd]_{63..32} \leftarrow (GPR[rs]_{63..32} - GPR[rt]_{63..32})_{31..0}$$
$$GPR[rd]_{95..64} \leftarrow (GPR[rs]_{95..64} - GPR[rt]_{95..64})_{31..0}$$
$$GPR[rd]_{127..96} \leftarrow (GPR[rs]_{127..96} - GPR[rt]_{127..96})_{31..0}$$

	127–96	95–64	63–32	31–0
rs	A3	A2	A1	A0
	–	–	–	–
rt	B3	B2	B1	B0
rd	A3–B3	A2–B2	A1–B1	A0–B0

Exceptions:

None

PXOR

Parallel Exclusive OR

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	PXOR 10011	MMI2 001001
6	5	5	5	5	6

C790

Format: PXOR rd, rs, rt

Purpose: To do a bitwise logical EXCLUSIVE OR.

Description: rd ← rs XOR rt

The contents of GPR *rs* are combined with the contents of GPR rt in a bitwise logical exclusive OR operation. The result is placed into GPR *rd*.

This instruction operates on 128-bit registers.

Operation:

$GPR[rd]_{127..0} \leftarrow GPR[rs]_{127..0}$ xor $GPR[rt]_{127..0}$

	127 64	63 0
rs	A1	A0
	XOR	XOR
rt	B1	B0
rd	A1 XOR B1	A0 XOR B0

Exceptions:

None

QFSRV Quadword Funnel Shift Right Variable QFSRV

31 26	25 21	20 16	15 11	10 6	5 0
MMI 011100	rs	rt	rd	QFSRV 11011	MMI1 101000
6	5	5	5	5	6

C790

Format: QFSRV rd, rs, rt

Purpose: To right shift a quadword by a variable number of bits.

Description: rd ← (rs, rt) >> SA

The content of GPR *rt* is concatenated with the content of GPR *rs* producing the intermediate result *rs:rt*. This value is shifted right by the number of bits specified in the shift amount register SA. The least significant 16 bytes (i.e. quadword) of the shifted result is placed into GPR *rd*.

Restriction:

Note that SA can be loaded only with byte shift values (MTSAB) or halfword shift values (MTSAH); i.e. with bit shift amounts that are multiples of 8 or 16.

This instruction operates on 128-bit registers.

Operation:

```
if ( SA == 0 ) then
    GPR[rd]127..0      ←  GPR[rt]127..0
else
    GPR[rd]127..0      ←  GPR[rs](SA–1)..0 || GPR[rt]127..SA
endif
```

Programming Note:

1. A left funnel shift by an amount of *s* bytes can be done by setting SA to 16-*s* using the MTSAB instruction, provided that *s* is not 0. Similarly, a left funnel shift by *s* halfwords can be done by setting SA to 8-*s* using the MTSAH instruction, provided that *s* is not 0. A quick way to perform this computation is as follows:

```
// Register %sal contains the left shift amount
subi   %samt, %sal, 1
mtsab%samt, -1

// Following QFSRV does a shift left by %sal bytes
qfsrv  %dst, %src1, %src2
```

2. QFSRV can be used to rotate a 128-bit quantity r by setting both source operands rs and rt to register r. For example, the following code sequence rotates right the value in wide register %5 by 3 halfwords(i.e. 48 bits), and deposits the result in wide register %6.

```
mtsah    %0, 3
qfsrv    %6, %5, %5
```

SQ

Store Quadword

SQ

31 26	25 21	20 16	15 0
SQ 011111	base	rt	offset
6	5	5	16

C790

Format: SQ rt, offset (base)

Purpose: To store a quadword to memory.

Description: memory [base + offset] ← rt

The 128-bit quadword in GPR *rt* is stored in memory at the location specified by the effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address. The least significant four bits of the effective address are masked to zero (effectively creating an aligned address) before being used to access memory. No address exceptions due to alignment are possible.

Restrictions:

The effective address doesn't have to be naturally aligned. The least significant 4 bits of the effective address are ignored.

Operation:

vAddr ← sign_extend (offset) + $GPR[base]_{31..0}$
$vAddr_{3..0} = 0^4$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
quadword ← $GPR[rt]_{127..0}$
StoreMemory (uncached, QUADWORD, quadword, pAddr, vAddr, DATA)

Exceptions:

TLB Refill
TLB Invalid
Address Error

Programming Notes:

None

B.5 C790-Specific Instruction Encoding

31..26	25..0
OpCode	

OpCode bits 28..26

Instructions encoded by **OpCode** field (MMI, LQ, SQ)

bits 31..29	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1 001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2 010	COP0	COP1	*	*	BEQL	BNEL	BLEZL	BGTZL
3 011	DADDI	DADDIU	LDL	LDR	***MMI***	*	LQ	SQ
4 100	LB	LH	LWL	LW	LBU	LHU	LWR	LWU
5 101	SB	SH	SWL	SW	SDL	SDR	SWR	CACHE
6 110	η	LWC1	η	PREF	η	LDC1	η	LD
7 111	η	SWC1	η	*	η	SDC1	η	SD

31..26	25..6	5..0
OpCode = MMI		**function**

function bits 2..0

Instructions encoded by **function** field when OpCode field = MMI

bits 5..3	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	MADD	MADDU	*	*	PLZCW	*	*	*
1 001	***MMI0*** δ	***MMI2*** δ	*	*	*	*	*	*
2 010	MFHI1	MTHI1	MFLO1	MTLO1	*	*	*	*
3 011	MULT1	MULTU1	DIV1	DIVU1	*	*	*	*
4 100	MADD1	MADDU1	*	*	*	*	*	*
5 101	***MMI1*** δ	***MMI3*** δ	*	*	*	*	*	*
6 110	PMFHL	PMTHL	*	*	PSLLH	*	PSRLH	PSRAH
7 111	*	*	*	*	PSLLW	*	PSRLW	PSRAW

31 26		10 6	5 0
OpCode = MMI		**function**	MMI0

function bits 7..6 Instructions encoded by **function** field when OpCode field = MMI & bit 5..0 = MMI0

bits 10..8	0 00	1 01	2 10	3 11
0 000	PADDW	PSUBW	PCGTW	PMAXW
1 001	PADDH	PSUBH	PCGTH	PMAXH
2 010	PADDB	PSUBB	PCGTB	*
3 011	*	*	*	*
4 100	PADDSW	PSUBSW	PEXTLW	PPACW
5 101	PADDSH	PSUBSH	PEXTLH	PPACH
6 110	PADDSB	PSUBSB	PEXTLB	PPACB
7 111	*	*	PEXT5	PPAC5

31 26		10 6	5 0
OpCode = MMI		**function**	MMI1

function bits 7..6 Instructions encoded by **function** field when OpCode field = MMI & bit 5..0 = MMI1

bits 10..8	0 00	1 01	2 10	3 11
0 000	*	PABSW	PCEQW	PMINW
1 001	PADSBH	PABSH	PCEQH	PMINH
2 010	*	*	PCEQB	*
3 011	*	*	*	*
4 100	PADDUW	PSUBUW	PEXTUW	*
5 101	PADDUH	PSUBUH	PEXTUH	*
6 110	PADDUB	PSUBUB	PEXTUB	QFSRV
7 111	*	*	*	*

31 26		10 6	5 0
OpCode = MMI		**function**	MMI2

function bits 7..6

Instructions encoded by **function** field when OpCode field = MMI & bit 5..0 = MMI2

bits 10..8	0 00	1 01	2 10	3 11
0 000	PMADDW	*	PSLLVW	PSRLVW
1 001	PMSUBW	*	*	*
2 010	PMFHI	PMFLO	PINTH	*
3 011	PMULTW	PDIVW	PCPYLD	*
4 100	PMADDH	PHMADH	PAND	PXOR
5 101	PMSUBH	PHMSBH	*	*
6 110	*	*	PEXEH	PREVH
7 111	PMULTH	PDIVBW	PEXEW	PROT3W

31 26		10 6	5 0
OpCode = MMI		**function**	MMI3

function bits 7..6

Instructions encoded by **function** field when OpCode field = MMI & bit 5..0 = MMI3

bits 10..8	0 00	1 01	2 10	3 11
0 000	PMADDUW	*	*	PSRAVW
1 001	*	*	*	*
2 010	PMTHI	PMTLO	PINTEH	*
3 011	PMULTUW	PDIVUW	PCPYUD	*
4 100	*	*	POR	PNOR
5 101	*	*	*	*
6 110	*	*	PEXCH	PCPYH
7 111	*	*	PEXCW	*

* This OpCode is reserved for future use. An attempt to execute it causes a Reserved Instruction exception.

δ This OpCode indicates an instruction class. The instruction word must be further decoded by examining additional tables that show the values for another instruction fields.

η This OpCode is reserved for one of the following instructions which are currently not supported: DMULT, DMULTU, DDIV, DDIVU, LL, LLD, SC, SCD, LWC2, SWC2. An attempt to execute it causes a Reserved Instruction exception.

C. COP0 System Control Coprocessor Instruction Set Details

This appendix provides a detailed description of the operation of each System Control Coprocessor (COP0) instruction.

COP0 instructions perform operations specifically on the System Control Coprocessor registers to manipulate the memory management and exception handing facilities of the processor.

COP0 Coprocessor instructions are enabled if the processor is in Kernel mode, *or* if bit 28 (CU[0]) is set in the *Status* register. Otherwise, executing one of these instructions generates a Coprocessor Unusable exception. The only exception to this rule are the EI and the DI instructions which *never* generate Coprocessor Unusable exceptions.

When the *EDI* bit in the *Status* register is set, the EI and DI instructions operate in User, Supervisor, and Kernel modes independent of whether COP0 coprocessor usable bit (*Status.CU[0]*) is set or not. When the EDI bit is cleared EI and DI work as NOPs in User and Supervisor modes independent of whether COP0 coprocessor usable bit (*Status.CU[0]*) is set or not, and executes properly in Kernel mode.

BC0F Branch on Coprocessor 0 False BC0F

31–26	25–21	20–16	15–0
COP0 010000	BC0 01000	BC0F 00000	offset
6	5	5	16

MIPS I

Format: BC0F offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and 16-bit *offset*, shifted left two bits and sign-extended. If coprocessor 0's condition signal, as sampled during the previous instruction, is false, then the program branches to the target address with a delay of one instruction.

Restrictions:

Because the coprocessor 0 condition is externally supplied, there is no way to synchronize the change/update of the condition and the execution of this instruction.

Operation:

```
I:      tgt_offset ← sign_extend (offset || 0²)
        condition ← not CPCOND0

I+1:    if condition then
                PC ← PC + tgt_offset
        endif
```

Exceptions:

Coprocessor Unusable exception

BC0FL Branch on Coprocessor 0 False Likely BC0FL

31 26	25 21	20 16	15 0
COP0 010000	BC0 01000	BC0FL 00010	offset
6	5	5	16

MIPS II

Format: BC0FL offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of coprocessor 0's condition signal, as sampled during the previous instruction, is false, the program branches to the target address with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Restrictions:

Because the coprocessor 0 condition is externally supplied, there is no way to synchronize the change/update of the condition and the execution of this instruction.

Operation:

```
I:      tgt_offset ← sign_extend (offset || 0^2)
        condition ← not CPCOND0

I+1:    if condition then
                PC ← PC + tgt_offset
        endif
```

Exceptions:

Coprocessor Unusable exception

BC0T Branch on Coprocessor 0 True BC0T

31–26	25–21	20–16	15–0
COP0 010000	BC0 01000	BC0T 00001	offset
6	5	5	16

MIPS I

Format: BC0T offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the coprocessor 0'z condition signal is true, then the program branches to the target address, with a delay of one instruction.

Restrictions:

Because the coprocessor 0 condition is externally supplied, there is no way to synchronize the change/update of the condition and the execution of this instruction.

Operation:

```
I:      tgt_offset ← sign_extend (offset || 0^2)
        condition ← not CPCOND0

I+1:    if condition then
                PC ← PC + tgt_offset
        endif
```

Exceptions:

Coprocessor Unusable exception

BC0TL Branch on Coprocessor 0 True Likely BC0TL

31–26	25–21	20–16	15–0
COP0 010000	BC0 01000	BC0TL 00011	offset
6	5	5	16

MIPS II

Format: BC0TL offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of coprocessor 0's condition signal, as sampled during the previous instruction, is true, the program branches to target address with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Restrictions:

Because the coprocessor 0 condition is externally supplied, there is no way to synchronize the change/update of the condition and the execution of this instruction.

Operation:

```
I:      tgt_offset ← sign_extend (offset || 0^2)
        condition ← not CPCOND0

I+1:    if condition then
                PC ← PC + tgt_offset
        else
                NullifyCurrentInstruction()
        endif
```

Exceptions:

Coprocessor Unusable exception

CACHE Cache CACHE

31 – 26	25 – 21	20 – 16	15 – 0
CACHE 101111	base	op (See table)	offset
6	5	5	16

R4000

Format: CACHE op, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address (VA). The VA is translated to a physical address (PA) through the memory management unit and its TLB, and the 5-bit OpCode (decode in the table below) specifies a cache operation for that address, together with the affected cache. Operation of this instruction on any combination not listed in the table below is undefined. The operation of this instruction on uncached and uncached accelerated addresses is also undefined unless it is index-type sub-operation.

Table C-1. CACHE Instruction Op Field Encoding

Mnemonic	OpCode	CACHE Instruction	Target
IXIN	00111	INDEX INVALIDATE	Instruction Cache
IXLTG	00000	INDEX LOAD TAG	Instruction Cache
IXSTG	00100	INDEX STORE TAG	Instruction Cache
IHIN	01011	HIT INVALIDATE	Instruction Cache
IFL	01110	FILL	Instruction Cache
IXLDT	00001	INDEX LOAD DATA	Instruction Cache
IXSDT	00101	INDEX STORE DATA	Instruction Cache
BXLBT	00010	INDEX LOAD BTAC	BTAC
BXSBT	00110	INDEX STORE BTAC	BTAC
BFH	01100	BTAC FLUSH	BTAC
BHINBT	01010	HIT INVALIDATE BTAC	BTAC
DXWBIN	10100	INDEX WRITE BACK INVALIDATE	Data Cache
DXLTG	10000	INDEX LOAD TAG	Data Cache
DXSTG	10010	INDEX STORE TAG	Data Cache
DXIN	10110	INDEX INVALIDATE	Data Cache
DHIN	11010	HIT INVALIDATE	Data Cache
DHWBIN	11000	HIT WRITEBACK INVALIDATE	Data Cache
DXLDT	10001	INDEX LOAD DATA	Data Cache
DXSDT	10011	INDEX STORE DATA	Data Cache
DHWOIN	11100	HIT WRITEBACK W/O INVALIDATE	Data Cache

Operation:

vAddr←$(offset_{15})^{16}$ || $offset_{15..0}$ + $GPR[base]_{31..0}$
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
CacheOp (op, vAddr, pAddr)

Exceptions:

Coprocessor Unusable exception
TLB Refill
TLB Invalid
Address Error

C.1.1 Notes on the CACHE Instruction Sub-operations

Cache Virtual Address

The CACHE instruction uses the following portions of the Virtual Address (VA) computed by adding the offset to the base to specify a cache block and way:

- VA[13:6] defines a 64-byte line in the data cache array
- VA[13:6] defines a 64-byte line in the instruction cache array
- In both cases, VA[0] defines the way needed by Index sub-operations

When accessing data in the caches, VA[13:2] is used to read or write a specific data word in the data cache and VA[13:2] is use to read or write a specific instruction in the instruction cache.

Cache Physical Address

The CACHE instruction computes the Physical Address (PA) to access memory for cache Hit Invalidate (I) and Fill (I) sub-operations in the following manner:

- VA[31:6] is computed from the CACHE instruction by adding the offset to the base and then the result is translated to produce PA[31:6]

The CACHE instruction computes the Physical Address (PA) to access memory for cache Hit Invalidate (D), Hit Writeback Invalidate (D), Hit Writeback Without Invalidate (D) sub-operations in the following manner:

- VA[31:6] is computed from the CACHE instruction by adding the offset to the base and then the result is translated to produce PA[31:6]

BTAC Virtual Address

The CACHE instruction uses the following portions of the Virtual Address (VA) computed by adding the offset to the base to check if there is an entry that matches the VA:

- VA[31:3] defines an entry in the BTAC

BTAC Index Bits

Since the BTAC is has 64 entries the VA[5:0] computed from the CACHE instruction by adding the offset to the base is used to index the BTAC.

COP0 Not Usable

If COP0 is not usable (if not in Kernel mode, Status.CU0 must be set for COP0 to be usable), a Coprocessor unusable exception is taken.

TLB Exceptions on Cache Operations

TLB Refill and TLB Invalid exceptions can occur only for the following sub-operations:

1. Hit Invalidate (I)
2. Fill (I)
3. Hit Invalidate (D)
4. Hit Writeback Invalidate (D)
5. Hit Writeback without Invalidate (D)

The TLB Modified exception is never generated.

Hit Sub-operation Accesses

A Hit sub-operation accesses the specified cache as a normal data reference, and performs the specified operation if the cache line contains valid data at the specified physical address (a hit). The operation is undefined if a CACHE sub-operation hit occurs in both ways of the cache.

Breakpoint Exception

Breakpoint exceptions can not be generated by any of the CACHE sub-operations (note that an Instruction Address Breakpoint can still be done on the CACHE instruction itself).

Address Error Exception

None of the CACHE sub-operations will generate an Address Error exception due to misalignment of the VA created by the CACHE instruction as described above. The following CACHE sub-operations can generate privilege-type Address Error exceptions:

1. Hit Invalidate (I)
2. Fill (I)
3. Hit Invalidate (D)
4. Hit Writeback Invalidate (D)
5. Hit Writeback without Invalidate (D)

C.1.2 Sub-Operation Descriptions

Note on Cache Enable Status

All Instruction cache related suboperations perform their function regardless of the value of the *ICE* bit of the *Config* register. (i.e., regardless of whether the Instruction cache is enabled or not.)

All data cache related suboperations perform their function regardless of the value of the *DCE* bit of the *Config* register. (i.e., regardless of whether the data cache is enabled or not.)

All BTAC-related suboperations perform their function regardless of the value of the BPE bit of the *Config* register.

Op = 00111 Index Invalidate (I)

Index Invalidate (I) sets a line in the instruction cache to Invalid. VA[13:6] defines the index of the line and VA[0] defines the way to be invalidated. The LRF bit does not change.

Op = 00000 Index Load Tag (I)

Index Load Tag (I) reads the instruction cache tag array fields into the COP0 TagLO register. VA[13:6] defines the index and VA[0] defines the way of the tag to be read. The following mapping defines the sub-operation:

- TagLO[4] = LRF bit
- TagLO[5] = VALID bit
- TagLO[31:12] = Tag[19:0]

All other TagLO bits are undefined.

Op = 00100 Index Store Tag (I)

Index Store Tag (I) stores the COP0 TagLO register into the instruction cache tag array. VA[13:6] defines the index and VA[0] defines the way of the tag to be read. The following mapping defines the sub-operation:

- LRF bit = TagLO[4]
- VALID bit = TagLO[5]
- Tag[19:0] = TagLO[31:12]

Note that it is perfectly feasible to invalidate the cache line using this sub-operation.

Op = 01011 Hit Invalidate (I)

Hit Invalidate (I) invalidates a line in the instruction cache which matches the PA[31:6] computed from the CACHE instruction. Both way tags at VA[13:6] are read from the instruction cache.

If the Valid bit of one of the entries is a 1 and the PA of the CACHE instruction matches the Tag from that entry of the instruction cache tag array, the Valid bit of the entry is changed to a 0 (Invalid). The LRF bit does not change. This sub-operation also invalidates BTAC entries which match VA[31:6].

Op = 01110 Fill (I)

Fill (I) brings in a cache line from memory and stores it in the instruction cache. The following sequence is followed:

1. The PA computed from the CACHE instruction is used to fetch the cache line from memory.
2. The line is loaded into the cache line addressed by VA[13:6] and the way of cache is defined by the rules of the LRF bits.
3. The corresponding instruction cache tag is loaded with the PFN and the entry is validated.

Op = 00001 Index Load Data (I)

Index Load Data (I) reads a single instruction from the instruction cache data array and stores it into the COP0 TagLO and TagHI registers. VA[13:2] defines the index and VA[0] defines the way of the instruction cache to be read. The following mapping defines the sub-operation:

- TagLO[31:0] = 32-bit instruction
- TagHI[3:0] = SteeringBits[3:0]
- TagHI[5:4] = BHT[1:0]

All other TagHI bits are undefined.

Op = 00101 Index Store Data (I)

Index Store Data (I) stores the COP0 TagLO and TagHI registers into the instruction cache data array.

VA[13:2] defines the index and VA[0] defines the way of the instruction cache to be written. The following mapping defines the sub-operation:

- 32-bit instruction = TagLO[31:0]
- SteeringBits[3:0] = TagHI[3:0]
- BHT[1:0] = TagHI[5:4]

The BHT[1:0] bits are associated with the instruction pair at VA[13:3]. This sub-operation invalidates all BTAC entries.

Op = 00010 Index Load BTAC (B)

Index Load BTAC (B) reads a single BTAC entry and stores it into the COP0 TagLO registers. VA[5:0] defines the index of the BTAC entry to be read. The following mapping defines the sub-operation:

- TagLO[0] = Valid Bit
- TagLO[31:3] = FetchAddress[28:0]
- TagHI[31:2] = TargetAddress[29:0]

All other TagLO and TagHI bits are undefined.

Op = 00110 Index Store BTAC (B)

Index Store BTAC (B) stores the COP0 TagLO and TagHI registers into a single BTAC entry. VA[5:0] defines the index of the BTAC entry to be written. The following mapping defines the sub-operation:

- Valid Bit = TagLO[0]
- FetchAddress[28:0] = TagLO[31:3]
- TargetAddress[29:0] = TagHI[31:2]

Op = 01100 BTAC Flush (B)

This sub-operation invalidates the complete BTAC by writing a 0 into the valid bits of all the entries of the BTAC.

Op = 01010 Hit Invalidate BTAC (B)

Hit Invalidate BTAC (B) invalidates an entry in the BTAC which matches the VA[31:3] computed from the CACHE instruction. If the VA[31:3] matches an entry in the BTAC and its Valid bit is equal to 1 then the Valid bit is changed to a 0. The result is undefined if there are plural of entries that matches the VA.

Op = 10100 Index Writeback Invalidate (D)

Index Writeback Invalidate (D) sub-operation sets a cache line in the data cache to Invalid and writes back any dirty data to the CPU bus. VA[13:6] defines the index and VA[0] defines the way of the data cache line to be invalidated. The invalidation takes place by writing a 0 to the Valid bit. The LRF bit does not change.

The PA where the cache line will be written to is calculated by appending VA[11:6] to the 20-bit PFN field from the data cache tag to form PA[31:6]. This address represents a cache line address.

Op = 10000 Index Load Tag (D)

Index Load Tag (D) reads the data cache tag array fields into the COP0 TagLO register. VA[13:6] defines the index and VA[0] defines the way of the tag to be read. The following mapping defines the sub-operation:

- TagLO[3] = Lock bit
- TagLO[4] = LRF bit
- TagLO[5] = Valid bit
- TagLO[6] = Dirty bit
- TagLO[31:12] = Tag[31:12]

All other TagLO bits are undefined.

Op = 10010 Index Store Tag (D)

Index Store Tag (D) stores the COP0 TagLO register into the data cache tag array. VA[13:6] defines the index and VA[0] defines the way of the tag to be written. The following mapping defines the sub-operation:

- Lock bit = TagLO[3]
- LRF bit = TagLO[4]
- Valid bit = TagLO[5]
- Dirty bit = TagLO[6] & TagLO[5]
- Tag[19:0] = TagLO[31:12]

Op = 10110 Index Invalidate (D)

Index Invalidate (D) sets a line in the data cache to Invalid. VA[13:6] defines the index of the line and VA[0] defines the way to be invalidated. The Lock bit, Dirty bit, and Valid bit are changed to zero. The LRF bit doesn't change.

Op = 11010 Hit Invalidate (D)

Hit Invalidate (D) invalidates an entry in the data cache which matches the PA computed from the CACHE instruction. Both way tags at VA[13:6] are read from the data cache.

If the Valid bit of the entry is one and the PA of the CACHE instruction matches the Tag from the data cache tag array, the Valid bit of the entry is changed to zero (Invalid). The Lock bit and Dirty bit are also changed to zero. The LRF bit does not change.

Op = 11000 Hit Writeback Invalidate (D)

Hit Writeback Invalidate (D) sub-operation invalidates an entry in the data cache which matches the PA computed from the CACHE instruction. Additionally it writes back any dirty data to the CPU bus. Both way tags at VA[13:6] are read from the data cache. The Lock bit, Dirty bit, and Valid bit are changed to zero. The LRF bits are not modified.

If the PA computed from the CACHE instruction matches the tag from the data cache tag array and the Valid bit is 1 then the Valid bit is changed to 0. Further more if the Dirty bit is 1 then the cache line is written to the physical address calculated by appending VA[11:6] to the 20-bit PFN field from the data cache tag to form PA[31:6]. This address represents a cache line physical address.

Op = 10001 Index Load Data (D)

Index Load Data (D) reads a single word from the data cache data array and stores it into the COP0 TagLO register. VA[13:2] defines the index and VA[0] defines the way of the data cache to be read. The following mapping defines the sub-operation:

- TagLO[31:0] = 32-bit data

Op = 10011 Index Store Data (D)

Index Store Data (D) stores the COP0 TagLO register into the data cache data array. VA[13:2] defines the index and VA[0] defines the way of the data cache to be written. The following mapping defines the sub-operation:

- 32-bit data = TagLO[31:0]

Op = 11100 Hit Writeback Without Invalidate (D)

Hit Writeback Without Invalidate (D) sub-operation writes back any dirty data to the CPU bus. Both way tags at VA[13:6] are read from the data cache. The Dirty bit is changed to zero. The LRF bits are not modified.

If the PA computed from the CACHE instruction matches the tag from the data cache tag array and the Valid and Dirty bits are 1 then the cache line is written to the physical address calculated by appending VA[11:6] to the 20-bit PFN field from the data cache tag to form PA[31:6]. This address represents a cache line physical address.

Programming Notes:

For all CACHE sub-operations which operate on the instruction cache the following programming restrictions have to be followed:

1. A sequence of CACHE instructions has to be directly preceded and followed by a SYNC.P instruction.
2. Each individual FILL sub-operation has to be followed by a SYNC.L instruction.

For all CACHE sub-operations which operate on the data cache the following programming restrictions have to be followed:

1. A sequence of CACHE instructions have to be directly preceded and followed by a SYNC.L instruction.
2. Each of the three WRITEBACK sub-operations have to be individually followed by a SYNC.L instruction.

For all CACHE sub-operations which operate on the BTAC the following programming restrictions have to be followed:

1. A sequence of CACHE instructions have to be directly preceded and followed by a SYNC.P instruction.

C.1.3 Updates of Data Tag Status Bits

The following table summarizes the updates of Data Tag status bits for various Cache sub-operations. The values in the table for Hit Writeback Invalidate, Hit Writeback Without Invalidate, and Hit Invalidate only apply if there is a hit in the data cache. If there is no hit, the status bits are unchanged.

Table C-2. Data Tag Status Bit Modifications

Cache Instruction	LRF Bit	Lock Bit	Dirty Bit	Valid Bit
Index Load Data	unchanged	unchanged	unchanged	unchanged
Index Store Data	unchanged	unchanged	unchanged	unchanged
Index Load Tag	unchanged	unchanged	unchanged	unchanged
Index Store Tag	loaded	loaded	loaded	loaded
Index Writeback Invalidate	unchanged	cleared	cleared	cleared
Index Invalidate	unchanged	cleared	cleared	cleared
Hit Invalidate	unchanged	cleared	cleared	cleared
Hit Writeback Invalidate	unchanged	cleared	cleared	cleared
Hit Writeback Without Invalidate	unchanged	unchanged	cleared	unchanged

DI — Disable Interrupt — DI

31 26	25 21	20 6	5 0
COP0 010000	C0 10000	0 000 0000 0000 0000	DI 111001
6	5	15	6

C790

Format: DI

Description:

DI instruction clears the *EIE* bit in the *Status* register and disable all interrupts (except NMI and SIO). When the *EIE* bit is cleared, all interrupts are disabled regardless of the value of *IE* bit in the *Status* register.

When the *EDI* bit in the *Status* register is set, the DI instruction operates in User, Supervisor, and Kernel modes independent of whether COP0 coprocessor usable bit (*Status.CU[0]*) is set or not. When this bit is cleared EI and DI work as NOPs in User and Supervisor modes independent of whether COP0 coprocessor usable bit (*Status.CU[0]*) is set or not, and executes properly in Kernel mode.

Operation:

If (Status.EDI = 1) || (Status.EXL = 1) || (Status.ERL = 1) || (Status.KSU = 00_2) then
    Status.EIE ← 0
endif

Exceptions:

None

EI Enable Interrupt EI

31–26	25–21	20–6	5–0
COP0 010000	C0 10000	0 000 0000 0000 0000	EI 111000
6	5	15	6

C790

Format: EI

Description:

EI instruction sets the *EIE* bit in the *Status* register. When the *EIE* bit is set, all interrupts are enabled if the *IE* bit in the *Status* register is 1, *EXL* bit is 0, and *ERL* bit is 0.

When the *EDI* bit in the Status register is set, the EI instruction operates in User, Supervisor, and Kernel modes independent of whether COP0 coprocessor usable bit (*Status.CU[0]*) is set or not. When this bit is cleared EI and DI work as NOPs in User and Supervisor modes independent of whether COP0 coprocessor usable bit (*Status.CU[0]*) is set or not, and executes properly in Kernel mode.

Operation:

If (Status.EDI = 1) || (Status.EXL = 1) || (Status.ERL = 1) || (Status.KSU = 00_2) then
    Status.EIE ← 1
endif

Exceptions:

None

ERET

Exception Return

31–26	25–21	20–6	5–0
COP0 010000	C0 10000	0 000 0000 0000 0000	ERET 011000
6	5	15	6

R4000

Format: ERET

Description:

ERET is the instruction for returning from an interrupt, exception, or error trap. Unlike a branch or jump instruction, ERET does not execute the next instruction.

ERET must not itself be placed in a branch delay slot.

If the processor is servicing a Level 2 exception, then load the PC from the *ErrorEPC* and clear the *ERL* bit of the *Status* register (bit 2 in *Status* register). Otherwise (*ERL* = 0), load the PC from the *EPC*, and clear the *EXL* bit of the *Status* register (bit 1 in *Status* register).

Operation:

```
if Status.ERL = 1 then
        PC ← ErrorEPC
        Status.ERL ← 0
else
        PC ← EPC
        Status.EXL ← 0
endif
```

Exceptions:

Coprocessor Unusable exception

Implementation Note:

ERET flushes the execution pipelines of the CPU before fetching the instruction from the target. Any pending loads, stores, ongoing multiplies, divides, multiply-accumulates and COP1 instructions are not flushed.

Programming Notes:

Any Reserved Instruction must not be placed in a branch delay slot just after ERET instruction. Please pay careful attention if any instruction is placed in the branch delay slot, because the instruction in the branch delay slot may be executed incompletely before flushing. It is commended that NOP is placed in the branch delay slot.

MFBPC Move from Breakpoint Control Register MFBPC

31–26	25–21	20–16	15–11	10–3	2–0
COP0 010000	MF0 00000	rt	Debug 11000	0 0000 0000	MFBPC 000
6	5	5	5	8	3

C790

Format: MFBPC rt

Description:

The contents of the *Breakpoint Control* register of the COP0 are loaded into general register *rt*.

Operation:

data ← CPR[0, Breakpoint Control]
GPR[rt] ← $(data_{31})^{32}$ || $data_{31..0}$

Exceptions:

Coprocessor Unusable exception

MFC0 Move from System Control Coprocessor MFC0

31 26	25 21	20 16	15 11	10 0
COP0 010000	MF0 00000	rt	rd	0 000 0000 0000
6	5	5	5	11

R4000

Format: MFC0 rt, rd

Description:

The contents of coprocessor register *rd* of the COP0 are loaded into general register *rt*.

Operation:

data ← CPR[0, rd]
GPR[rt] ← $(data_{31})^{32}$ || $data_{31..0}$

Exceptions:

Coprocessor Unusable exception

MFDAB Move from Data Address Breakpoint register MFDAB

31–26	25–21	20–16	15–11	10–3	2–0
COP0 010000	MF0 00000	rt	Debug 11000	0 0000 0000	MFDAB 100
6	5	5	5	8	3

C790

Format: MFDAB rt

Description:

The contents of *Data Address Breakpoint* register of the COP0 are loaded into general register *rt*.

Operation:

data ← CPR[0, Data Address Breakpoint]
GPR[rt] ← $(data_{31})^{32}$ || $data_{31..0}$

Exceptions:

Coprocessor Unusable exception

MFDABM — Move from Data Address Breakpoint Mask Register — MFDABM

31–26	25–21	20–16	15–11	10–3	2–0
COP0 010000	MF0 00000	rt	Debug 11000	0 0000 0000	MFDABM 101
6	5	5	5	8	3

C790

Format: MFDABM rt

Description:

The contents of *Data Address Breakpoint Mask* register of the COP0 are loaded into general register *rt*.

Operation:

data ← CPR[0, Data Address Breakpoint Mask]
GPR[rt] ← $(data_{31})^{32}$ || $data_{31..0}$

Exceptions:

Coprocessor Unusable exception

MFDVB Move from Data value Breakpoint Register MFDVB

31 26	25 21	20 16	15 11	10 3	2 0
COP0 010000	MF0 00000	rt	Debug 11000	0 0000 0000	MFDVB 110
6	5	5	5	8	3

C790

Format: MFDVB rt

Description:

The contents of *Data Value Breakpoint* register of the COP0 are loaded into general register *rt*.

Operation:

data ← CPR[0, Data Value Breakpoint]
GPR[rt] ← $(data_{31})^{32}$ || $data_{31..0}$

Exceptions:

Coprocessor Unusable exception

MFDVBM Move from Data Value Breakpoint Mask Register MFDVBM

31–26	25–21	20–16	15–11	10–3	2–0
COP0 010000	MF0 00000	rt	Debug 11000	0 0000 0000	MFDVBM 111
6	5	5	5	8	3

C790

Format: MFDVBM rt

Description:

The contents of *Data Value Breakpoint Mask* register of the COP0 are loaded into general register *rt*.

Operation:

data ← CPR[0, Data Value Breakpoint Mask]
GPR[rt] ← $(data_{31})^{32}$ || $data_{31..0}$

Exceptions:

Coprocessor Unusable exception

MFIAB Move from Instruction Address Breakpoint Register MFIAB

31 26	25 21	20 16	15 11	10 3	2 0
COP0 010000	MF0 00000	rt	Debug 11000	0 0000 0000	MFIAB 010
6	5	5	5	8	3

C790

Format: MFIAB rt

Description:

The contents of *Instruction Address Breakpoint* register of the COP0 are loaded into general register *rt*.

Operation:

data ← CPR[0, Instruction Address Breakpoint]
GPR[rt] ← $(data_{31})^{32}$ || $data_{31..0}$

Exceptions:

Coprocessor Unusable exception

MFIABM — Move from Instruction Address Breakpoint Mask Register — MFIABM

31–26	25–21	20–16	15–11	10–3	2–0
COP0 010000	MF0 00000	rt	Debug 11000	0 0000 0000	MFIABM 011
6	5	5	5	8	3

C790

Format: MFIABM rt

Description:

The contents of *Instruction Address Breakpoint Mask* register of the COP0 are loaded into general register *rt*.

Operation:

data ← CPR[0, Instruction Address Breakpoint Mask]
GPR[rt] ← $(data_{31})^{32}$ || $data_{31..0}$

Exceptions:

Coprocessor Unusable exception

MFPC Move from Performance Counter MFPC

31 26	25 21	20 16	15 11	10 6	5 1	0
COP0 010000	MF0 00000	rt	Perf 11001	0 00000	reg	1
6	5	5	5	5	5	1

C790

Format: MFPC rt, reg

Description:

The contents of *Performance Counter* register of the COP0 are loaded into general register *rt*.

The reg OpCode bit indicates the number of *Performance Counters.* Only register 0 and 1 are valid in the C790 implementation.

Operation:

data ← CPR[0, Performance Counter (reg)]
GPR[rt] ← $(data_{31})^{32}$ || $data_{31..0}$

Exceptions:

Coprocessor Unusable exception

MFPS Move from Performance Event Specifier MFPS

31–26	25–21	20–16	15–11	10–6	5–1	0
COP0 010000	MF0 00000	rt	Perf 11001	0 00000	reg	0
6	5	5	5	5	5	1

C790

Format: MFPS rt, reg

Description:

The contents of *Performance Control* register of the COP0 are loaded into general register *rt*.

The reg OpCode bit indicates the number of *Performance Counter Control* registers. Only register 0 is valid in the C790 implementation.

Operation:

data ← CPR[0, Performance Control (reg)]
GPR[rt] ← $(data_{31})^{32}$ || $data_{31..0}$

Exceptions:

Coprocessor Unusable exception

MTBPC Move to Breakpoint Control Register MTBPC

31 26	25 21	20 16	15 11	10 3	2 0
COP0 010000	MT0 00100	rt	Debug 11000	0 0000 0000	MTBPC 000
6	5	5	5	8	3

C790

Format: MTBPC rt

Description:

The contents of general register *rt* are loaded into *Breakpoint Control* register of COP0.

Operation:

data ← GPR[rt]
CPR[0, Breakpoint Control] ← data

Programming Notes:

All MTBPC instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

Exceptions:

Coprocessor Unusable exception

MTC0 — Move to System Control Coprocessor — MTC0

31–26	25–21	20–16	15–11	10–0
COP0 010000	MT0 00100	rt	rd	0 000 0000 0000
6	5	5	5	11

R4000

Format: MTC0 rt, rd

Description:

The contents of general register *rt* are loaded into coprocessor register *rd* of COP0.

Operation:

data ← GPR[rt]
CPR[0, rd] ← data

Programming Notes:

1. All MTC0 instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update. There is one exception to this rule:
   a) An MTC0 instruction which loads the EntryHi COP0 register can be followed by a TLBWI or a TLBWR instruction without having an intervening SYNC.P instruction. This special case is handled by a hardware interlock.
2. It is required that the MTC0 instruction to EntryHi register MUST be executed either from unmapped space or from global mapped space (mapped space with a TLB entry which has the G bit set). Furthermore, the BTAC is flushed whenever the EntryHi register is updated.
3. Modifying *CONFIG.K0* via a MTC0 instruction should not occur from kseg0 space.
4. A SYNC.L instruction is needed before executing a MTC0 instruction which modifies *CONFIG.NBE* or *CONFIG.DCE.*
5. Updating the performance counter registers via a MTC0 instruction while the performance counters are enabled will result in undefined counter values.

Exceptions:

Coprocessor Unusable exception

MTDAB Move to Data Address Breakpoint Register MTDAB

31–26	25–21	20–16	15–11	10–3	2–0
COP0 010000	MT0 00100	rt	Debug 11000	0 0000 0000	MTDAB 100
6	5	5	5	8	3

C790

Format: MTDAB rt

Description:

The contents of general register *rt* are loaded into *Data Address Breakpoint* register of COP0.

Operation:

data ← GPR[rt]
CPR[0, Data Address Breakpoint] ← data

Programming Notes:

All MTDAB instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

Exceptions:

Coprocessor Unusable exception

MTDABM — Move to Data Address Breakpoint Mask Register — MTDABM

31–26	25–21	20–16	15–11	10–3	2–0
COP0 010000	MT0 00100	rt	Debug 11000	0 0000 0000	MTDABM 101
6	5	5	5	8	3

C790

Format MTDABM rt

Description:

The contents of general register *rt* are loaded into *Data Address Breakpoint Mask* register of COP0.

Operation:

data ← GPR[rt]
CPR[0, Data Address Breakpoint Mask] ← data

Programming Notes:

All MTDABM instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

Exceptions:

Coprocessor Unusable exception

MTDVB Move to Data Value Breakpoint Register MTDVB

31–26	25–21	20–16	15–11	10–3	2–0
COP0 010000	MT0 00100	rt	Debug 11000	0 0000 0000	MTDVB 110
6	5	5	5	8	3

C790

Format: MTDVB rt

Description:

The contents of general register *rt* are loaded into *Data Value Breakpoint* register of COP0.

Operation:

data ← GPR[rt]
CPR[0, Data Value Breakpoint] ← data

Programming Notes:

All MTDVB instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

Exceptions:

Coprocessor Unusable exception

MTDVBM — Move to Data Value Breakpoint Mask Register — MTDVBM

31–26	25–21	20–16	15–11	10–3	2–0
COP0 010000	MT0 00100	rt	Debug 11000	0 0000 0000	MTDVBM 111
6	5	5	5	8	3

C790

Format: MTDVBM rt

Description:

The contents of general register *rt* are loaded into *Data Value Breakpoint Mask* register of COP0.

Operation:

data ← GPR[rt]
CPR[0, Data Value Breakpoint Mask] ← data

Programming Notes:

All MTDVBM instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

Exceptions:

Coprocessor Unusable exception

MTIAB

Move to Instruction Address Breakpoint Register

31–26	25–21	20–16	15–11	10–3	2–0
COP0 010000	MT0 00100	rt	Debug 11000	0 0000 0000	MTIAB 010
6	5	5	5	8	3

C790

Format: MTIAB rt

Description:

The contents of general register *rt* are loaded into *Instruction Address Breakpoint* register of COP0.

Operation:

data ← GPR[rt]
CPR[0, Instruction Address Breakpoint] ← data

Programming Notes:

All MTIAB instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

Exceptions:

Coprocessor Unusable exception

MTIABM — Move to Instruction Address Mask Breakpoint Register — MTIABM

31–26	25–21	20–16	15–11	10–3	2–0
COP0 010000	MT0 00100	rt	Debug 11000	0 0000 0000	MTIABM 011
6	5	5	5	8	3

C790

Format: MTIABM rt

Description:

The contents of general register *rt* are loaded into *Instruction Address Mask Breakpoint* register of COP0.

Operation:

data ← GPR[rt]
CPR[0, Instruction Address Mask Breakpoint] ← data

Programming Notes:

All MTIABM instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

Exceptions:

Coprocessor Unusable exception

MTPC Move to Performance Counter MTPC

31 26	25 21	20 16	15 11	10 6	5 1	0
COP0 010000	MT0 00100	rt	Perf 11001	0 00000	reg	1
6	5	5	5	5	5	1

C790

Format: MTPC rt, reg

Description:

The contents of general register *rt* are loaded into *Performance Counter* register.

The *reg* OpCode bit indicates the number of *Performance Counters.* Only register 0 and 1 are valid in the C790 implementation.

Operation:

data ← GPR[rt]
CPR[0, Performance Counter (reg)] ← data

Programming Notes:

All MTPC instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

Updating the performance counters via a MTPC instruction while the performance counters are enabled will result in undefined counter values.

Exceptions:

Coprocessor unusable exception

MTPS Move to Performance Event Specifier MTPS

31–26	25–21	20–16	15–11	10–6	5–1	0
COP0 010000	MT0 00100	rt	Perf 11001	0 00000	reg	0
6	5	5	5	5	5	1

C790

Format: MTPS rt, reg

Description:

The contents of general register *rt* are loaded into *Performance Control* register.

The *reg* OpCode bit indicates the number of *Performance Control* registers. Only register 0 is valid in the C790 implementation.

Operation:

data ← GPR[rt]
CPR[0, Performance Control (reg)] ← data

Programming Notes:

All MTPS instructions MUST be followed by a SYNC.P instruction as a barrier to guarantee COP0 register update.

Exceptions:

Coprocessor unusable exception

TLBP   Probe TLB for Matching Entry   TLBP

31–26	25–21	20–6	5–0
COP0 010000	C0 10000	0 000 0000 0000 0000	TLBP 001000
6	5	15	6

R4000

Format: TLBP

Description:

The *Index* register is loaded with the address of the TLB entry whose contents match the contents of the *EntryHi* register. If no TLB entry matches, the high-order bit of the *Index* register is set to 1. Note that the virtual address in the *EntryHi* register is masked with the corresponding *mask* field of the TLB entry prior to the comparison.

The architecture does not specify the operation of memory references associated with the instruction immediately after a TLBP instruction, nor is the operation specified if more than one TLB entry matches.

Operation:

Index $\leftarrow 1 \,||\, 0^{25} \,||\, \text{undefined}^{6}$
for i in 0..TLBEnteries-1
    if $(\text{TLB}[i]_{95..77} = ((\text{not TLB}[i]_{127..109})$ and $\text{EntryHi}_{31..13}))$ and $(\text{TLB}[i]_{76}$ or
        $(\text{TLB}[i]_{71..64} = \text{EntryHi}_{7..0}))$ then
            Index $\leftarrow 0^{26} \,||\, i_{5..0}$
    endif
endfor

Programming Notes:

The TLBP instruction MUST be immediately followed by SYNC.P or ERET instruction

Exceptions:

Coprocessor Unusable exception

TLBR — Read Indexed TLB Entry — TLBR

31–26	25–21	20–6	5–0
COP0 010000	C0 10000	0 000 0000 0000 0000	TLBR 000001
6	5	15	6

R4000

Format: TLBR

Description:

The *EntryHi*, *EntryLo*, and PageMask registers are loaded with the contents of the TLB entry pointed at by the contents of the TLB *Index* register.

The *G* bit (which controls ASID matching) read from the TLB is written into both of the *EntryLo0* and *EntryLo1* registers. Depending the value in PageMask register used for a TLB write instruction, the value read out from TLB may not retrieve what was originally written. See Description for TLBWI/TLBWR instruction.

Operation:

$PageMask \leftarrow TLB[Index_{5..0}]_{127..96}$
$EntryHi \leftarrow (TLB[Index_{5..0}]_{95..77} \,\|\, 0^5 \,\|\, TLB[Index_{5..0}]_{71..64})$ and (not $TLB[Index_{5..0}]_{127..96}$)
$EntryLo0 \leftarrow TLB[Index_{5..0}]_{63..33} \,\|\, TLB[Index_{5..0}]_{76}$
$EntryLo1 \leftarrow TLB[Index_{5..0}]_{31..1} \,\|\, TLB[Index_{5..0}]_{76}$

Programming Notes:

The TLBR instruction MUST be executed from either unmapped space or global mapped space (mapped space with a TLB entry which has the G bit set).

The TLBR instruction MUST be immediately followed by SYNC.P or ERET instruction.

Exceptions:

Coprocessor Unusable exception

TLBWI Write Index TLB Entry TLBWI

31 26	25 21	20 6	5 0
COP0 010000	C0 10000	0 000 0000 0000 0000	TLBWI 000010
6	5	15	6

R4000

Format: TLBWI

Description:

The TLB entry pointed at by the contents of the TLB *Index* register is loaded with the contents of the *PageMask, EntryHi, EntryLo0* and *EntryLo1* registers.

The *G* bit of the TLB is written with the logical AND of the *G* bits in the *EntryLo0* and *EntryLo1* registers. The virtual address in the *EntryHi* register is modified by the *Mask* field of the *PageMask* register before being written into the TLB.

The operation is invalid (and the results are unspecified) if contents of the TLB *Index* register are greater than the number of TLB entries in the processor.

In the C790 processor, a TLB write instruction is used to write the whole page frame number from the *EntryLo* registers to the TLB entry. Depending on the page size specified in the corresponding *PageMask* register, the lower bits of PFN may not be used for address translation and lower bits of VPN2 in EntryHi register which is masked by the content of PageMask register are forced to zeros during a TLB write. This does not affect TLB address translation, however, a TLB read may not retrieve what was originally written.

Operation:

$TLB[Index_{5..0}] \leftarrow$
$PageMask \parallel ((EntryHi_{31..13} \parallel (EntryLo0_0 \text{ and } EntryLo1_0) \parallel EntryHi_{11..0}) \text{ and }$
$(\text{not } PageMask)) \parallel EntryLo0_{31..1} \parallel 0 \parallel EntryLo1_{31..1} \parallel 0$

Programming Notes:

The TLBWI instruction MUST be executed from either unmapped space or global mapped space (mapped space with a TLB entry which has the G bit set).

The TLBWI instruction MUST be followed by a ERET or a SYNC.P instruction to insure TLB update.

Exceptions:

Coprocessor Unusable exception

TLBWR — Write Random TLB Entry — TLBWR

31–26	25–21	20–6	5–0
COP0 010000	C0 10000	0 000 0000 0000 0000	TLBWR 000110
6	5	15	6

R4000

Format: TLBWR

Description:

The TLB entry pointed at by the contents of the TLB *Random* register is loaded with the contents of the *PageMask*, *EntryHi*, *EntryLo0* and *EntryLo1* registers.

The G bit of the TLB is written with the logical AND of the *G* bits in the *EntryLo0* and *EntryLo1* registers. The virtual address in the *EntryHi* register is modified by the *Mask* field of the *PageMask* register before being written into the TLB.

In the C790 processor, a TLB write instruction is used to write the whole page frame number from the *EntryLo* registers to the TLB entry. Depending on the page size specified in the corresponding *PageMask* register, the lower bits of PFN may not be used for address translation and lower bits of VPN2 in EntryHi register which is masked by the content of PageMask register are forced to zeros during a TLB write. This does not affect TLB address translation, however, a TLB read may not retrieve what was originally written.

Operation:

$TLB[Random_{5..0}] \leftarrow$
$PageMask \,||\, ((EntryHi_{31..13} \,||\, (EntryLo0_0 \text{ and } EntryLo1_0) \,||\, EntryHi_{11..0}) \text{ and }$
$(\text{not } PageMask)) \,||\, EntryLo0_{31..1} \,||\, 0 \,||\, EntryLo1_{31..1} \,||\, 0$

Programming Notes:

The TLBWR instruction MUST be executed from either unmapped space or global mapped space (mapped space with a TLB entry which has the G bit set).

The TLBWR instruction MUST be followed by a ERET or a SYNC.P instruction to insure TLB update.

Exceptions:

Coprocessor Unusable exception

C.2 COP0 Instruction Encoding

31 26	25 0
OpCode	

OpCode bits 28..26 — Instructions encoded by **OpCode** field (COP0, CACHE)

bits 31..29	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1 001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2 010	***COP0*** δ	COP1	*	*	BEQL	BNEL	BLEZL	BGTZL
3 011	DADDI	DADDIU	LDL	LDR	MMI	*	LQ	SQ
4 100	LB	LH	LWL	LW	LBU	LHU	LWR	LWU
5 101	SB	SH	SWL	SW	SDL	SDR	SWR	CACHE
6 110	η	LWC1	η	PREF	η	LDC1	η	LD
7 111	η	SWC1	η	*	η	SDC1	η	SD

31 26	25 21	20 0
OpCode = COP0	**rs**	

rs bits 23..21 — Instructions encoded by **rs** field when OpCode field = COP0

bits 25..24	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 00	MF0	*	*	*	MT0	*	*	*
1 01	***BC0*** δ	*	*	*	*	*	*	*
2 10	***C0*** δ	*	*	*	*	*	*	*
3 11	*	*	*	*	*	*	*	*

31 26	25 21	20 16	15 11	10 3	2 0
OpCode = COP0	**rs =** MF0 or MT0		**rd =** Debug•		**function**

function bits 2..0 — Instructions encoded by **function** field when OpCode field = COP0 & rd field = Debug

rs field	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
MF0	MFBPC	φ	MFIAB	MFIABM	MFDAB	MFDABM	MFDVB	MFDVBM
MT0	MTBPC	φ	MTIAB	MTIABM	MTDAB	MTDABM	MTDVB	MTDVBM

31 26	25 21	20 16	15 11	10 1	0
OpCode = COP0	**rs =** MF0 or MT0		**rd** = Perf•		**function**

function bits 0 — Instructions encoded by **function** field when OpCode field = COP0 & rd field = Perf

rs field	0	1
MF0	MFPS	MFPC
MT0	MTPS	MTPC

• Debug and Perf are the CP0 register names.
Debug = 11000 (24), Perf = 11001 (25)

31–26	25–21	20–16	15–0
OpCode = COP0	**rs =** *BC0*	**rt**	

rt bits 18..16 — Instructions encoded by **rt** field when OpCode field = COP0 & rs field = BC0

bits 20..19	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 00	BC0F	BC0T	BC0FL	BC0TL	*	*	*	*
1 01	*	*	*	*	*	*	*	*
2 10	*	*	*	*	*	*	*	*
3 11	*	*	*	*	*	*	*	*

31–26	25–21	20–6	5–0
OpCode = COP0	**rs =** *C0*		**function**

function bits 2..0 — Instructions encoded by **function** field when OpCode field = COP0 & rs field = C0

bits 5..3	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	φ	TLBR	TLBWI	φ	φ	φ	TLBWR	φ
1 001	TLBP	φ	φ	φ	φ	φ	φ	φ
2 010	φ	φ	φ	φ	φ	φ	φ	φ
3 011	ERET	φ	φ	φ	φ	φ	φ	φ
4 100	φ	φ	φ	φ	φ	φ	φ	φ
5 101	φ	φ	φ	φ	φ	φ	φ	φ
6 110	φ	φ	φ	φ	φ	φ	φ	φ
7 111	EI	DI	φ	φ	φ	φ	φ	φ

* This OpCode is reserved for future use. An attempt to execute it causes a Reserved Instruction exception.

φ This OpCode is reserved for future use. An attempt to execute it produces an undefined result. The result may be a Reserved Instruction exception but this is not guaranteed.

δ This OpCode indicates an instruction class. The instruction word must be further decoded by examining additional tables that show the values for another instruction field.

η This OpCode is reserved for one of the following instructions which are currently not supported: DMULT, DMULTU, DDIV, DDIVU, LL, LLD, SC, SCD, LWC2, SWC2. An attempt to execute it causes a Reserved Instruction exception.

D. COP1 (FPU) Instruction Set Details

This appendix provides a detailed description of each of the COP1 coprocessor instructions. COP1 is implemented as a floating point unit (FPU).

The instruction descriptions provide:

- a bit by bit field definition of the instruction word signifying that instruction
- a verbal description of the operation performed by the instruction
- pseudo-code identifying the entire sphere of influence of the instruction in terms of operand dependency and the state (s) of the processor changed.

Omission of any/all states is taken to mean that the same have not changed by the act of execution of the instruction under description.

D.1 Conventions Used in This Chapter

D.1.1 Instruction Description Notation and Functions

The *Operation* sections of the instruction descriptions use a high-level language notation, or pseudocode, to describe the instruction's operations. Symbols, functions, and structures used in the *Operation* sections are described here.

The notation *FPR* as used here refers to the 32 floating-point registers *FPR0* through *FPR31* of the FPU.

D.1.2 Pseudocode Language Statement Execution

Each of the high-level language statements in an operation description is executed in sequential order (as modified by conditional and loop constructs).

D.1.3 Pseudocode Symbols

Special symbols used in the notation are described in Appendix A.

D.2 Definitions for Pseudocode Functions Used in Operation Descriptions

A variety of functions are used in the pseudocode descriptions to make the pseudocode more readable and also to abstract implementation-specific behavior. These functions are defined in Appendix A; in addition, certain COP1 FPU-specific functions are described in the following section. The following pseudocode notation is used in functions in the descriptions of floating-point operations:

Pseudocode Function	Meaning
StoreFPR (fpr, value)	FPR[fpr] ← value
ConvertFmt (value, fmt1, fmt2)	The value in the format fmt1 is converted to a value in the format fmt2.
Negate (value)	The value is negated by changing the sign bit value.
Sign-extend (Value)	A sign-extended 32-bit value has bits 63..31 of equal value

D.3 Instruction Descriptions

Descriptions of FPU Instructions follow.

ABS.fmt

Floating Point Absolute Value

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	ABS 000101
6	5	5	5	5	6

MIPS I

Format: ABS.S fd, fs

ABS.D fd, fs

Purpose: To compute the absolute value of an FP value.

Description: fd ← absolute (fs)

The absolute value of the value in FPR *fs* is placed in FPR *fd*. The operand and result are values in format *fmt*.

This operation is arithmetic; a NaN operand signals invalid operation.

Restrictions:

The field *fs* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, fmt, AbsoluteValue (ValueFPR (fs, fmt)))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Unimplemented Operation
- Invalid Operation

ADD.fmt

Floating Point Add

31–26	25–21	20–16	15–11	10–6	5–0
COP1 010001	fmt	ft	fs	fd	ADD 000000
6	5	5	5	5	6

MIPS I

Format: ADD.S fd, fs, ft

ADD.D fd, fs, ft

Purpose: To add FP values.

Description: fd ← fs + ft

The value in FPR *ft* is added to the value in FPR *fs*. The result is calculated to infinite precision, rounded according to the current rounding mode in FCR31, and placed into FPR *fd*. The operands and result are values in format *fmt*.

Restrictions:

The field *fs, ft* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, fmt, ValueFPR (fs, fmt) + ValueFPR (ft, fmt))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Unimplemented Operation
- Invalid Operation
- Inexact
- Overflow
- Underflow

BC1F — Branch on FP False — BC1F

31–26	25–21	20–16	15–0
COP1 010001	BC1 01000	BC1F 00000	offset
6	5	5	16

MIPS I

Format: BC1F offset

Purpose: To test an FP condition code and do a PC-relative conditional branch.

Description: if (C = 0) then branch where C is $FCR31_{23}$

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the result of the last floating point compare is false, branch to the effective target address after the instruction in the delay slot is executed.

An FP condition code is set by the FP compare instruction, *C.cond.fmt*.

Operation:

```
I:       condition ← (FCR31_23 = 0)
         target_offset ← (offset_15)^(GPRLEN-(16+2)) || offset || 0^2
I+1:     if condition then
               PC ← PC + target
         endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

BC1T Branch on FP True BC1T

31 26	25 21	20 16	15 0
COP1 010001	BC1 01000	BC1T 00001	offset
6	5	5	16

MIPS I

Format: BC1T offset

Purpose: To test an FP condition code and do a PC-relative conditional branch.

Description: if (C = 1) then branch where C is $FCR31_{23}$.

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the result of the last floating point compare is true, branch to the effective target address after the instruction in the delay slot is executed.

An FP condition code is set by the FP compare instruction, *C.cond.fmt*.

Operation:

```
I:      condition ← (FCR31_23 = 1)
        target ← (offset_15)^(GPRLEN-(16+2)) || offset || 0^2
I+1:    if condition then
            PC ← PC + target
        endif
```

Exceptions:

Coprocessor Unusable
Reserved Instruction

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128KB. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

C.cond.fmt Floating Point Compare C.cond.fmt

31–26	25–21	20–16	15–11	10–6	5–4	3–0
COP1 010001	fmt	ft	fs	0 00000	FC 11	cond
6	5	5	5	5	2	4

MIPS I

Format: C.cond.S fs, ft

C.cond.D fs, ft

Purpose: To compare FP values and record the Boolean result in a condition code.

Description: C ← fs *compare_cond* ft

The value in FPR *fs* is compared to the value in FPR *ft*; the values are in format *fmt*. The comparison is exact and neither overflows nor underflows. If the comparison specified by *cond* $_{2..1}$ is true for the operand values, then the result is true, otherwise it is false. If no exception is taken, the result is written into condition code C; true is 1 and false is 0.

If *cond* $_3$ is set and at least one of the values is a NaN, an Invalid Operation condition is raised; the result depends on the FP exception model currently active.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written and an Invalid Operation exception is taken immediately. Otherwise, the Boolean result is written into condition code C

There are four mutually exclusive ordering relations for comparing floating-point values; one relation is always true and the others are false. The familiar relations are *greater than*, *less than*, and *equal*. In addition, the IEEE floating-point standard defines the relation *unordered* which is true when at least one operand value is NaN; NaN compares unordered with everything, including itself. Comparisons ignore the sign of zero, so +0 equals -0.

The comparison condition is a logical predicate, or equation, of the ordering relations such as “less than or equal”, “equal”, “not less than”, or “unordered or equal”. Compare distinguishes sixteen comparison predicates. The Boolean result of the instruction is obtained by substituting the Boolean value of each ordering relation for the two FP values into equation. If the *equal* relation is true, for example, then all four example predicates above would yield a true result. If the *unordered* relation is true then only the final predicate, “unordered or equal” would yield a true result.

Logical negation of a compare result allows eight distinct comparisons to test for sixteen predicates as shown in Table D-1. Each mnemonic tests for both a predicate and its logical negation. For each mnemonic, compare tests the truth of the first predicate. When the first predicate is true, the result is true as shown in the “if predicate is true” column (note that the False predicate is never true and False/True do not follow the normal pattern). When the first predicate is true, the second predicate must be false, and vice versa. The truth of the second predicate is the logical negation of the instruction result. After a compare instruction, test for the truth of the first predicate with the Branch on FP True (BC1T) instruction and the truth of the second with Branch on FP False (BC1F).

Table D-1. FPU Comparisons Without Special Operand Exceptions

Instr	Comparison Predicate					Comparison CC Result		Instr	
cond Mne-monic	name of predicate and logically negated predicate (abbreviation)	relation values				If pred-icate is true	Inv Op excp if Q NaN	cond field	
		>	<	=	?			3	2..0
F	False [this predicate is always False, it never has a True result] True (T)	F T	F T	F T	F T	F	No	0	0
UN	Unordered Ordered (OR)	F T	F T	F T	T F	T F			1
EQ	Equal Not Equal (NEQ)	F T	F T	T F	F T	T F			2
UEQ	Unordered or Equal Ordered or Greater than or Less than (OGL)	F T	F T	T F	T F	T F			3
OLT	Ordered or Less Than Unordered or Greater than or Equal (UGE)	F T	T F	F T	F T	T F			4
ULT	Unordered or Less Than Ordered or Greater than or Equal (OGE)	F T	T F	F T	T F	T F			5
OLE	Ordered or Less than or Equal Unordered or Greater Than (UGT)	F T	T F	T F	F T	T F			6
ULE	Unordered or Less than or Equal Ordered or Greater Than (OGT)	F T	T F	T F	T F	T F			7
key: "?" = unordered, ">" = greater than, "<" = less than, "=" is equal, "T" = True, "F" = False									

There is another set of eight compare operations, distinguished by a *cond*$_3$ value of 1, testing the same sixteen conditions. For these additional comparisons, if at least one of the operands is a NaN, including Quiet NaN, then an Invalid Operation condition is raised. If the Invalid Operation condition is enabled in the FCR31, then an Invalid Operation exception occurs.

Table D-2 FPU Comparisons With Special Operand Exceptions for QNaNs

Instr	Comparison Predicate					Comparison CC Result		Instr	
cond Mne-monic	name of predicate and logically negated predicate (abbreviation)	relation values >	<	=	?	If pred-icate is true	Inv Op excp if Q NaN	cond field 3	2..0
SF	Signaling False [this predicate Signaling True (ST) always False]	F T	F T	F T	F T	F	Yes	1	0
NGLE	Not Greater than or Less than or Equal Greater than or Less than or Equal (GLE)	F T	F T	F T	T F	T F			1
SEQ	Signaling Equal Signaling Not Equal (SNE)	F T	F T	T F	F T	T F			2
NGL	Not Greater than or Less than Greater than or Less than (GL)	F T	F T	T F	T F	T F			3
LT	Less Than Not Less Than (NLT)	F T	T F	F T	F T	T F			4
NGE	Not Greater than or Equal Greater than or Equal (GE)	F T	T F	F T	T F	T F			5
LE	Less than or Equal Not Less than or Equal (NLE)	F T	T F	T F	F T	T F			6
NGT	Not Greater Than Greater Than (GT)	F T	T F	T F	T F	T F			7
key: "?" = unordered, ">" = greater than, "<" = less than, "=" is equal, "T" = True, "F" = False									

Restrictions:

The field *fs* and *ft* must specify FPRs valid for operands of type *fmt*; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

```
if NaN (Value FPR (fs, fmt)) or NaN (ValueFPR (ft, fmt)) then
    less ← false
    equal ← false
    unordered ← true
    if t then
            SignalException (InvalidOperation)
    endif
else
    less ← ValueFPR (fs, fmt) < ValueFPR (ft, fmt)
    equal ← ValueFPR (fs, fmt) = ValueFPR (ft, fmt)
    unordered ← false
endif
condition ← (cond2 and less) or (cond1 and equal) or (cond0 and unordered)
C ← condition
```

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
    Unimplemented Operation
    Invalid Operation

Programming Notes:

FP computational instructions, including compare, that receive an operand value of Signaling NaN, will raise the Invalid Operation condition. The comparisons that raise the Invalid Operation condition for Quiet NaNs in addition to SNaNs, permit a simpler programming model if NaNs are errors. Using these compares, programs do not need explicit code to check for QNaNs causing the *unordered* relation. Instead, they take an exception and allow the exception handling system to deal with the error when it occurs. For example, consider a comparison in which we want to know if two numbers are equal, but for which unordered would be an error.

```
# comparisons using explicit tests for QNaN
    c.eq.d $f2,$f4 # check for equal
    nop
    bc1t L2 # it is equal
    c.un.d $f2,$f4 # it is not equal, but might be unordered
    bc1t ERROR# unordered goes off to an error handler
# not-equal-case code here
    ...
# equal-case code here
L2:
# -----------------------------------------------------------
# comparison using comparisons that signal QNaN
    c.seq.d $f2,$f4 # check for equal
    nop
    bc1t L2 # it is equal
    nop
# it is not unordered here...
# not-equal-case code here
    ...
#equal-case code here
L2:
```

CEIL.L.fmt Floating-Point Ceiling Convert to Long Fixed-Point CEIL.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CEIL.L 001010
6	5	5	5	5	6

MIPS III

Format: CEIL.L.S fd, fs

CEIL.L.D fd, fs

Purpose: To convert an FP value to 64-bit fixed-point, rounding up.

Description: fd ← convert_and_round (fs)

The value in FPR *fs* in format *fm*t, is converted to a value in 64-bit long fixed-point format rounding toward +∞ (rounding mode 2). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}-1$, is written to *fd*.

Restrictions:

The fields *fs and fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed-point; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
    Invalid Operation
    Unimplemented Operation
    Inexact
    Overflow

CEIL.W.fmt Floating-Point Ceiling Convert to Word Fixed-Point CEIL.W.fmt

31–26	25–21	20–16	15–11	10–6	5–0
COP1 010001	fmt	0 00000	fs	fd	CEIL.W 001110
6	5	5	5	5	6

MIPS II

Format: CEIL.W.S fd, fs

CEIL.W.D fd, fs

Purpose: To convert an FP value to 32-bit fixed-point, rounding up.

Description: fd ← convert_and_round (fs)

The value in FPR *fs* in format *fmt*, is converted to a value in 32-bit word fixed-point format rounding toward +∞ (rounding mode 2). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to $2^{31}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}-1$, is written to *fd*.

Restrictions:

The fields *fs and fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed-point; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Invalid Operation
- Unimplemented Operation
- Inexact
- Overflow

CFC1 Move Control Word from Floating Point CFC1

31 - 26	25 - 21	20 - 16	15 - 11	10 - 0
COP1 010001	CFC1 00010	rt	fs	0 000 0000 0000
6	5	5	5	11

MIPS I

Format: CFC1 rt, fs

Purpose: To copy a word from an FPU control register to a GPR.

Description: rt ← FP_Control[fs]

Copy the 32-bit word from FP (coprocessor 1) control register *fs* into GPR *rt*, sign-extending it if the GPR is 64 bits.

Restrictions:

There are only a couple control registers defined for the floating point unit. The result is not defined if *fs* specifies a register that does not exist.

Operation:

GPR[rt] ← sign_extend (FCR[fs])

Exceptions:

Coprocessor Unusable

CTC1 Move Control Word to Floating Point CTC1

31 26	25 21	20 16	15 11	10 0
COP1 010001	CTC1 00110	rt	fs	0 000 0000 0000
6	5	5	5	11

MIPS I

Format: CTC1 rt, fs

Purpose: To copy a word from a GPR to an FPU control register.

Description: FP_Control[fs] ← rt

Copy the low word from GPR *rt* into FP (coprocessor 1) control register *fs*.

Writing to control register 31, the *Floating-Point Control and Status Register* or FCR31, causes the appropriate exception if any cause bit and its corresponding enable bit are both set. The register will be written before the exception occurs.

Restrictions:

There are only a couple control registers defined for the floating point unit. The result is not defined if *fs* specifies a register that does not exist.

Operation:

temp ← $GPR[rt]_{31..0}$
FCR[fs] ← temp

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Invalid Operation
- Unimplemented Operation
- Inexact
- Overflow
- Underflow
- Division by Zero

CVT.D.fmt Floating-Point Convert to Double Foating Point CVT.D.fmt

31–26	25–21	20–16	15–11	10–6	5–0
COP1 010001	fmt	0 00000	fs	fd	CVT.D 100001
6	5	5	5	5	6

MIPS I, III

Format: CVT.D.S fd, fs

CVT.D.W fd, fs

CVT.D.L fd, fs

Purpose: To convert an FP or fixed-point value to double FP.

Description: fd ← convert_and_round (fs)

The value in FPR *fs* in format *fmt* is converted to a value in double floating-point format rounded according to the current rounding mode in FCR31. The result is placed in FPR *fd*.

If *fmt* is S or W, then the operation is always exact.

Restrictions:

The field *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for double floating point; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, D, ConvertFmt (ValueFPR (fs, fmt), fmt, D))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
  Invalid Operation
  Unimplemented Operation
  Inexact

Note:

Overflow and Underflow exceptions never occur because double precision data format can represent any value in other data types.

CVT.L.fmt — Floating-Point Convert to Long Fixed-Point — CVT.L.fmt

31–26	25–21	20–16	15–11	10–6	5–0
COP1 010001	fmt	0 00000	fs	fd	CVT.L 100101
6	5	5	5	5	6

MIPS III

Format: CVT.L.S fd, fs

CVT.L.D fd, fs

Purpose: To convert an FP value to a 64-bit fixed-point.

Description: fd ← convert_and_round (fs)

Convert the value in format *fmt* in FPR *fs* to long fixed-point format, round according to the current rounding mode in FCR31, and place the result in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}-1$, is written to *fd*.

Restrictions:

The field *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long floating point; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Invalid Operation
- Unimplemented Operation
- Inexact
- Overflow

CVT.S.fmt Floating-Point Convert to Single Floating-Point CVT.S.fmt

31–26	25–21	20–16	15–11	10–6	5–0
COP1 010001	fmt	0 00000	fs	fd	CVT.S 100000
6	5	5	5	5	6

MIPS I, III

Format: CVT.S.D fd, fs

CVT.S.W fd, fs

CVT.S.L fd, fs

Purpose: To convert an FP or fixed-point value to single FP.

Description: fd ← convert_and_round (fs)

The value in FPR *fs* in format *fmt* is converted to a value in single floating-point format rounded according to the current rounding mode in FCR31. The result is placed in FPR *fd*.

Restrictions:

The field *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for single floating point; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, S, ConvertFmt (ValueFPR (fs, fmt), fmt, S))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Invalid Operation
- Unimplemented Operation
- Inexact
- Overflow
- Underflow

CVT.W.fmt Floating-Point Convert to Word Fixed-Point CVT.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT.W 100100
6	5	5	5	5	6

MIPS I

Format: CVT.W.S fd, fs

CVT.W.D fd, fs

Purpose: To convert an FP value to a 32-bit fixed-point.

Description: fd ← convert_and_round (fs)

The value in FPR *fs* in format *fmt* is converted to a value in 32-bit word fixed-point format rounded according to the current rounding mode in FCR31. The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to $2^{31}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}-1$, is written to *fd*.

Restrictions:

The field *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed point; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Invalid Operation
- Unimplemented Operation
- Inexact
- Overflow

DIV.fmt

Floating Point Divide

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	DIV 000011
6	5	5	5	5	6

MIPS I

Format: DIV.S fd, fs, ft

DIV.D fd, fs, ft

Purpose: To divide FP values.

Description: fd ← fs / ft

The value in FPR *fs* is divided by the value in FPR *ft*. The result is calculated to infinite precision, rounded according to the current rounding mode in FCR31, and placed into FPR *fd*. The operands and result are values in format *fm*t.

Restrictions:

The field *fs, ft* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, fmt, ValueFPR (fs, fmt) / ValueFPR (ft, fmt))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Inexact
- Unimplemented Operation
- Division-by-zero
- Invalid Operation
- Overflow
- Underflow

DMFC1 Doubleword Move From Floating-Point DMFC1

31–26	25–21	20–16	15–11	10–0
COP1 010001	DMFC1 00001	rt	fs	0 000 0000 0000
6	5	5	5	11

MIPS III

Format: DMFC1 rt, fs

Purpose: To copy a doubleword from an FPR to a GPR.

Description: rt ← fs

The doubleword contents of FPR *fs* are placed into GPR *r*t.

If the coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR *fs* is held in an even/odd register pair. The low word is taken from the even register *fs* and the high word is from *f*s+1.

Restrictions:

If *fs* does not specify an FPR that can contain a doubleword, the result is undefined; see Floating Point Registers on page 10-2.

Operation:

```
if SizeFGR() = 64 then                /* 64-bit wide FGRs */
    data ← FGR[fs]
elseif fs0 = 0 then                   /* valid specifier, 32-bit wide FGRs */
    data ← FGR[fs+1] || FGR[fs]
else                                  /* undefined for odd 32-bit FGRs */
    UndefinedResult()
endif
GPR[rt] ← data
```

Exceptions:

Reserved Instruction
Coprocessor Unusable

DMTC1 Doubleword Move To Floating-Point DMTC1

31–26	25–21	20–16	15–11	10–0
COP1 010001	DMTC1 00101	rt	fs	0 000 0000 0000
6	5	5	5	11

MIPS III

Format: DMTC1 rt, fs

Purpose: To copy a doubleword from a GPR to an FPR.

Description: fs ← rt

The doubleword contents of GPR *rt* are placed into FPR *fs*.

If the coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR *fs* is held in an even/odd register pair. The low word is Placed in the even register *fs* and the high word is placed in *f*s+1.

Restrictions:

If *fs* does not specify an FPR that can contain a doubleword, the result is undefined; see Floating Point Registers on page 10-2.

Operation:

```
data ← GPR[rt]
if SizeFGR() = 64 then        /* 64-bit wide FGRs */
    FGR[fs] ← data
elseif fs0 = 0 then           /* valid specifier, 32-bit wide FGRs */
    FGR[fs+1] ← data63..32
    FGR[fs] ← data31..0
else                          /* undefined result for odd 32-bit FGRs */
    UndefinedResult()
endif
```

Exceptions:

Reserved Instruction
Coprocessor Unusable

FLOOR.L.fmt Floating-Point Floor Convert to Long Fixed-Point FLOOR.L.fmt

31–26	25–21	20–16	15–11	10–6	5–0
COP1 010001	fmt	0 00000	fs	fd	FLOOR.L 001011
6	5	5	5	5	6

MIPS III

Format: FLOOR.L.S fd, fs

FLOOR.L.D fd, fs

Purpose: To convert an FP value to a 64-bit fixed-point, rounding down.

Description: fd ← convert_and_round (fs)

The value in FPR *fs* in format *fm*t, is converted to a value in 64-bit long fixed-point format rounding toward –∞ (rounding mode 3). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}-1$, is written to *fd*.

Restrictions:

The field *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed point; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Invalid Operation
- Unimplemented Operation
- Inexact
- Overflow

FLOOR.W.fmt Floating-Point Floor Convert to Word Fixed-Point FLOOR.W.fmt

31–26	25–21	20–16	15–11	10–6	5–0
COP1 010001	fmt	0 00000	fs	fd	FLOOR.W 001111
6	5	5	5	5	6

MIPS II

Format: FLOOR.W.S fd, fs

FLOOR.W.D fd, fs

Purpose: To convert an FP value to a 32-bit fixed-point, rounding down.

Description: fd ← convert_and_round (fs)

The value in FPR *fs* in format *fm*t, is converted to a value in 32-bit word fixed-point format rounding toward –∞ (rounding mode 3). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to $2^{31}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}-1$, is written to *fd*.

Restrictions:

The field *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed point; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Invalid Operation
- Unimplemented Operation
- Inexact
- Overflow

LDC1 — Load Doubleword to Floating-Point — LDC1

31–26	25–21	20–16	15–0
LDC1 110101	base	ft	offset
6	5	5	16

MIPS II

Format: LDC1 ft, offset (base)

Purpose: To load a doubleword from memory to an FPR.

Description: ft ← memory[base+offset]

The contents of the 64-bit doubleword at the memory location specified by the aligned effective address are fetched and placed in FPR *ft*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

If coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR *ft* is held in an even/odd register pair. The low word is placed in the even register *ft* and the high word is placed in *ft*+1.

Restrictions:

If *ft* does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Resisters on page 10-2.

An Address Error exception occurs if $EffectiveAddress_{2..0} \neq 0$ (not doubleword-aligned).

Operation:

```
vAddr ← sign_extend (offset) + GPR[base]
if vAddr2..0 ≠ 0^3 then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
data ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
if SizeFGR() = 64 then              /* 64-bit wide FGRs */
    FGR[ft] ← data
elseif ft0 = 0 then                 /* valid specifier, 32-bit wide FGRs */
    FGR[ft+1] ← data63..32
    FGR[ft] ← data31..0
else                                /* undefined result for odd 32-bit FGRs */
    UndefinedResult()
endif
```

Exceptions:

Coprocessor Unusable
TLB Refill
TLB Invalid
Address Error

LWC1 Load Word to Floating Point LWC1

31 26	25 21	20 16	15 0
LWC1 110001	base	ft	offset
6	5	5	16

MIPS I

Format: LWC1 ft, offset (base)

Purpose: To load a word from memory to an FPR.

Description: ft ← memory[base+offset]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and placed into the low word of coprocessor 1 general register *ft* . The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

If coprocessor 1 general registers are 64-bits wide, bits 63..32 of register *ft* become undefined. See Floating Point Register on page 10-2.

Restrictions:

An Address Error exception occurs if $EffectiveAddress_{1..0} \neq 0$ (not word-aligned).

Operation: 32-bit Processors

I: /* "mem" is aligned 64-bits from memory. Pick out correct bytes. */
vAddr ← sign_extend (offset) + GPR[base]
if $vAddr_{1..0} \neq 0^2$ then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
I + 1: FGR[ft] ← mem

Operation: 64-bit Processors

/* "mem" is aligned 64-bits from memory. Pick out correct bytes. */
vAddr ← sign_extend (offset) + GPR[base]
if $vAddr_{1..0} \neq 0^2$ then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || 0^2))
mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
bytesel ← $vAddr_{2..0}$ xor (BigEndianCPU || 0^2)
if SizeFGR() = 64 then /* 64-bit wide FGRs */
FGR[ft] ← $undefined^{32}$ || $mem_{31+8*bytesel..8*bytesel}$
else /* 32-bit wide FGRs */
FGR[ft] ← $mem_{31+8*bytesel..8*bytesel}$
endif

Exceptions:

Coprocessor unusable
TLB Refill
TLB Invalid
Address Error

MFC1 — Move Word from Floating Point — MFC1

31–26	25–21	20–16	15–11	10–0
COP1 010001	MFC1 00000	rt	fs	0 000 0000 0000
6	5	5	5	11

MIPS I

Format: MFC1 rt, fs

Purpose: To copy a word from an FPU (COP1) general register to a GPR.

Description: rt ← fs

The low word from FPR *fs* is placed into the low word of GPR *rt*. If GPR *rt* is 64 bits wide, then the value is sign extended. See Floating Point Resisters on page 10-2.

Restrictions:

None

Operation:

GPR[rt] ← sign_extend (FPR[fs]$_{31..0}$)

Exceptions:

Coprocessor Unusable

MOV.fmt Floating Point Move MOV.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	MOV 000110
6	5	5	5	5	6

MIPS I

Format: MOV.S fd, fs

MOV.D fd, fs

Purpose: To move an FP value between FPRs.

Description: fd ← fs

The value in FPR *fs* is placed into FPR *fd* . The source and destination are values in format *fmt*.

The move is non-arithmetic; it causes no IEEE 754 exceptions.

Restrictions:

The field *fs* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, fmt, ValueFPR (fs, fmt))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
Unimplemented Operation

MTC1 Move Word to Floating Point MTC1

31–26	25–21	20–16	15–11	10–0
COP1 010001	MTC1 00100	rt	fs	0 000 0000 0000
6	5	5	5	11

MIPS I

Format: MTC1 rt, fs

Purpose: To copy a word from a GPR to an FPU (COP1) general register.

Description: fs ← rt

The low word in GPR *rt* is placed into the low word of floating-point (coprocessor 1) general register *fs*. If coprocessor 1 general registers are 64-bits wide, bits 63..32 of register *fs* become undefined. See Floating-Point Registers on page 10-2.

Operation:

```
data ← GPR[rt]31..0
if SizeFGR() = 64 then              /* 64-bit wide FGRs */
    FGR[fs] ← undefined^32 || data
else                                /* 32-bit wide FGRs */
    FGR[fs] ← data
endif
```

Exceptions:

Coprocessor Unusable

MUL.fmt

Floating Point Multiply

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	MUL 000010
6	5	5	5	5	6

MIPS I

Format: MUL.S fd, fs, ft

MUL.D fd, fs, ft

Purpose: To multiply FP values.

Description: fd ← fs × ft

The value in FPR *fs* is multiplied by the value in FPR *ft*. The result is calculated to infinite precision, rounded according to the current rounding mode in FCR31, and placed into FPR *fd*. The operands and result are value in format *fmt*.

Restrictions:

The field *fs, ft* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, fmt, ValueFPR (fs, fmt) * ValueFPR (ft, fmt))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
    Inexact
    Unimplemented Operation
    Invalid Operation
    Overflow
    Underflow

NEG.fmt Floating Point Negate NEG.fmt

31–26	25–21	20–16	15–11	10–6	5–0
COP1 010001	fmt	0 00000	fs	fd	NEG 000111
6	5	5	5	5	6

MIPS I

Format: NEG.S fd, fs

NEG.D fd, fs

Purpose: To negate a floating-point value.

Description: fd ← -(fs)

The value in FPR *fs* is negated and placed into FPR *fd*. The value is negated by changing the sign bit value. The operand and result are values in format *fmt*.

This operation is arthmetic; a NaN operand signals invalid operation.

Restrictions:

The field *fs* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, fmt, Negate (ValueFPR (fs, fmt))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
  Unimplemented Operation
  Invalid Operation

ROUND.L.fmt Floating Point Round to Long Fixed-Point ROUND.L.fmt

31–26	25–21	20–16	15–11	10–6	5–0
COP1 010001	fmt	0 00000	fs	fd	ROUND.L 001000
6	5	5	5	5	6

MIPS III

Format: ROUND.L.S fd, fs

ROUND.L.D fd, fs

Purpose: To convert an FP value to 64-bit fixed-point, round to nearest.

Description: fd ← convert_and_round (fs)

The value in FPR *fs* in format *fm*t, is converted to a value in 64-bit long fixed-point format rounding to nearest/even (rounding mode 0). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}-1$, is written to *fd*.

Restrictions:

The field *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed point; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt,L)

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
  Inexact
  Unimplemented Operation
  Overflow
  Invalid Operation

ROUND.W.fmt Floating Point Round to Word Fixed-Point ROUND.W.fmt

31–26	25–21	20–16	15–11	10–6	5–0
COP1 010001	fmt	0 00000	fs	fd	ROUND.W 001100
6	5	5	5	5	6

MIPS II

Format: ROUND.W.S fd, fs

ROUND.W.D fd, fs

Purpose: To convert an FP value to 32-bit fixed-point, round to nearest.

Description: fd ← convert_and_round (fs)

The value in FPR *fs* in format *fmt*, is converted to a value in 32-bit word fixed-point format rounding to nearest/even (rounding mode 0). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to $2^{31}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}-1$, is written to *fd*.

Restrictions:

The field *fs* and *fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed point; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt,W)

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Inexact
- Unimplemented Operation
- Overflow
- Invalid Operation

SDC1 Store Doubleword to Floating-Point SDC1

31 26	25 21	20 16	15 0
SDC1 111101	base	ft	offset
6	5	5	16

MIPS II

Format: SDC1 ft, offset (base)

Purpose: To store a doubleword from an FPR to memory.

Description: memory[base+offset] ← ft

The 64-bit doubleword in FPR *ft* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

If coprocessor 1 general registers are 32-bits wide (a native 32-bit processor or 32-bit register emulation mode in a 64-bit processor), FPR *ft* is held in an even/odd register pair. The low word is taken from the even register *ft* and the high word is from *ft*+1.

Restrictions:

If *ft* does not specify an FPR that can contain a doubleword, the result is undefined; see Floating-Point Resisters on page 10-2.

An Address Error exception occurs if $EffectiveAddress_{2..0} \neq 0$ (not doubleword-aligned).

Operation:

```
vAddr ← sign_extend (offset) + GPR[base]
if vAddr2..0 ≠ 0^3 then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
if SizeFGR() = 64 then              /* 64-bit wide FGRs */
    data ← FGR[ft]
elseif ft0 = 0 then                 /* valid specifier, 32-bit wide FGRs */
    data ← FGR[ft+1] || FGR[ft]
else                                /* undefined for odd 32-bit FGRs */
    UndefinedResult()
endif
StoreMemory(uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor Unusable
TLB Refill
TLB Invalid
TLB Modified
Address Error

SQRT.fmt Floating Point Square Root SQRT.fmt

31–26	25–21	20–16	15–11	10–6	5–0
COP1 010001	fmt	0 00000	fs	fd	SQRT 000100
6	5	5	5	5	6

MIPS II

Format: SQRT.S fd, fs

SQRT.D fd, fs

Purpose: To compute the square root of an FP value.

Description: fd ← SQRT (fs)

The square root of the value in FPR *fs* is calculated to infinite precision, rounded according to the current rounding mode in FCR31, and placed into FPR *fd*. The operand and result are values in format *fmt*.

If the value in FPR *fs* corresponds to –0, the result will be –0.

Restrictions:

If the value in FPR *fs* is less than 0, an Invalid Operation condition is raised.

The field *fs* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, fmt, SquareRoot (FPR (fs, fmt)))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Inexact
- Unimplemented Operation
- Invalid Operation

SUB.fmt Floating Point Subtract SUB.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	SUB 000001
6	5	5	5	5	6

MIPS I

Format: SUB.S fd, fs, ft

SUB.S fd, fs, ft

Purpose: To subtract FP values.

Description: fd ← fs - ft

The value in FPR *ft* is subtracted from the value in FPR *fs*. The result is calculated to infinite precision, rounded according to the current rounding mode in FCR31, and placed into FPR *fd* . The operands and result are value in format *fmt*.

Restrictions:

The field *fs, ft,* and *fd* must specify FPRs valid for operands of type *fmt*; see Floating-Point Resisters on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, fmt, ValueFPR (fs, fmt) – ValueFPR (ft, fmt))

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
  Inexact
  Unimplemented Operation
  Invalid Operation
  Overflow
  Underflow

SWC1 — Store Word from Floating Point — SWC1

31–26	25–21	20–16	15–0
SWC1 111001	base	ft	offset
6	5	5	16

MIPS I

Format: SWC1 ft, offset (base)

Purpose: To store a word from an FPR to memory.

Description: memory[base+offset] ← ft

The low 32-bit word from FPR *ft* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

An Address Error exception occurs if $EffectiveAddress_{1..0} \neq 0$ (not word-aligned).

Operation: 32-bit Processors

```
vAddr ← sign_extend (offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
data ← FGR[ft]
StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Operation: 64-bit Processors

```
vAddr ← sign_extend (offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then SignalException (AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr PSIZE-1..3 || (pAddr2..0 xor (ReverseEndian || 0^2 ))
bytesel ← vAddr2..0 xor (BigEndianCPU || 0^2 )
/* the bytes of the word are moved into the correct byte lanes */
if SizeFGR() = 64 then                          /* 64-bit wide FGRs */
  data ← 0^(32-8*bytesel) || FGR[ft]31..0 || 0^(8*bytesel) /* top or bottom wd of 64-bit data */
else                                            /* 32-bit wide FGRs */
  data ← 0^(32-8*bytesel) || FGR[ft] || 0^(8*bytesel) /* top or bottom wd of 64-bit data */
endif
StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor Unusable
TLB Refill
TLB Invalid
TLB Modified
Address Error

TRUNC.L.fmt Floating Point Truncate to Long Fixed-Point TRUNC.L.fmt

31–26	25–21	20–16	15–11	10–6	5–0
COP1 010001	fmt	0 00000	fs	fd	TRUNC.L 001001
6	5	5	5	5	6

MIPS III

Format: TRUNC.L.S fd, fs

TRUNC.L.D fd, fs

Purpose: To convert an FP value to 64-bit fixed-point, rounding toward zero.

Description: fd ← convert_and_round (fs)

The value in FPR *fs* in format *fm*t, is converted to a value in 64-bit long fixed-point format rounding toward zero (rounding mode 1). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{63} to $2^{63}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{63}-1$, is written to *fd*.

Restrictions:

The fields *fs and fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for long fixed-point; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L)

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Invalid Operation
- Unimplemented Operation
- Inexact
- Overflow

TRUNC.W.fmt Floating Point Truncate to Word Fixed-Point TRUNC.W.fmt

31–26	25–21	20–16	15–11	10–6	5–0
COP1 010001	ft	0 00000	fs	fd	TRUNC.W 001101
6	5	5	5	5	6

MIPS II

Format: TRUNC.W.S fd, fs

TRUNC.W.D fd, fs

Purpose: To convert an FP value to 32-bit fixed-point, rounding toward zero.

Description: fd ← convert_and_round (fs)

The value in FPR *fs* in format *fmt*, is converted to a value in 32-bit word fixed-point format rounding toward zero (rounding mode 1). The result is placed in FPR *fd*.

When the source value is Infinity, NaN, or rounds to an integer outside the range -2^{31} to $2^{31}-1$, the result cannot be represented correctly and an IEEE Invalid Operation condition exists.

The Invalid Operation flag is set in the FCR31. If the Invalid Operation enable bit is set in the FCR31, no result is written to *fd* and an Invalid Operation exception is taken immediately. Otherwise, the default result, $2^{31}-1$, is written to *fd*.

Restrictions:

The fields *fs and fd* must specify valid FPRs; *fs* for type *fmt* and *fd* for word fixed-point; see Floating-Point Registers on page 10-2. If they are not valid, the result is undefined.

Operation:

StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W)

Exceptions:

Coprocessor Unusable
Reserved Instruction
Floating-Point
- Invalid Operation
- Unimplemented Operation
- Inexact
- Overflow

D.4 COP1 Instruction Encoding

31 26	0
OpCode	

OpCode bits 28..26 Instructions encoded by **OpCode** field (COP1, LWC1, SWC1, LDC1, SDC1)

bits 31..29	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1 001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2 010	COP0	***COP1*** δ	*	*	BEQL	BNEL	BLEZL	BGTZL
3 011	DADDI	DADDIU	LDL	LDR	MMI	*	LQ	SQ
4 100	LB	LH	LWL	LW	LBU	LHU	LWR	LWU
5 101	SB	SH	SWL	SW	SDL	SDR	SWR	CACHE
6 110	η	LWC1	η	PREF	η	LDC1	η	LD
7 111	η	SWC1	η	*	η	SDC1	η	SD

31 26	25 21	0
OpCode = COP1	**rs**	

rs bits 23..21 Instructions encoded by **rs** field when OpCode field = COP1

bits 25..24	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 00	MFC1	DMFC1	CFC1	*	MTC1	DMTC1	CTC1	*
1 01	***BC1*** δ	*	*	*	*	*	*	*
2 10	***S*** δ	***D*** δ	φ	φ	***W*** δ	***L*** δ	φ	φ
3 11	φ	φ	φ	φ	φ	φ	φ	φ

31 26	25 21	20 16	0
OpCode = COP1	**rs =** *BC1*	**rt**	

rt bits 18..16 Instructions encoded by **rt** field
when OpCode field = COP1 & rs field = BC1

bits 20..19	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 00	BC1F	BC1T	*	*	*	*	*	*
1 01	*	*	*	*	*	*	*	*
2 10	*	*	*	*	*	*	*	*
3 11	*	*	*	*	*	*	*	*

31 26	25 21				5 0
OpCode = COP1	**rs =** *S, D*				**function**

function bits 2..0

Instructions encoded by **function** field
when OpCode field = COP1 & rs field = S, D

bits 5.3	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1 001	ROUND.L	TRUNC.L	CEIL.L	FLOOR.L	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2 010	φ	φ	φ	φ	φ	φ	φ	φ
3 011	φ	φ	φ	φ	φ	φ	φ	φ
4 100	CVT.S	CVT.D	φ	φ	CVT.W	CVT.L	φ	φ
5 101	φ	φ	φ	φ	φ	φ	φ	φ
6 110	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
7 111	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

31 26	25 21				5 0
OpCode = COP1	**rs =** *W, L*				**function**

function bits 2..0

Instructions encoded by **function** field
when OpCode field = COP1 & rs field = W, L

bits 5.3	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	φ	φ	φ	φ	φ	φ	φ	φ
1 001	φ	φ	φ	φ	φ	φ	φ	φ
2 010	φ	φ	φ	φ	φ	φ	φ	φ
3 011	φ	φ	φ	φ	φ	φ	φ	φ
4 100	CVT.S	CVT.D	φ	φ	φ	φ	φ	φ
5 101	φ	φ	φ	φ	φ	φ	φ	φ
6 110	φ	φ	φ	φ	φ	φ	φ	φ
7 111	φ	φ	φ	φ	φ	φ	φ	φ

* This OpCode is reserved for future use. An attempt to execute it causes a Reserved Instruction exception but this is not guaranteed.

φ This OpCode is reserved for future use. An attempt to execute it produces an undefined result. The result may be an Unimplemented Operation exception.

δ This OpCode indicates an instruction class. The instruction word must be further decoded by examining additional tables that show the values for another instruction field.

η This OpCode is reserved for one of the following instructions which are currently not supported: DMULT, DMULTU, DDIV, DDIVU, LL, LLD, SC, SCD, LWC2, SWC2. An attempt to execute it causes a Reserved Instruction exception.